

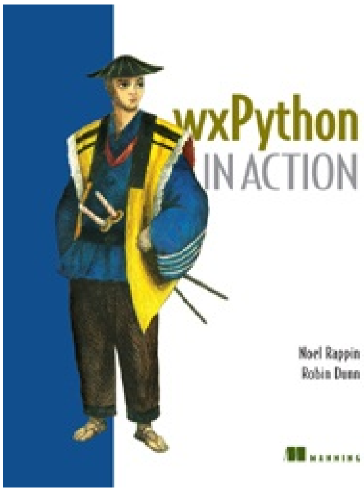
wxPython IN ACTION

Noel Rappin
Robin Dunn

MANNING

Table of Contents

活学活用wxPython



项目说明

- 本书英文版版权由原作者所有，前言及本书内容介绍两章由CliffPeng(Hairui)翻译，其他章节由ZZJ翻译完成，经CliffPeng整理成wiki。
- 中文译文为译者学习笔记性质，但非经译者允许请勿转载及摘录用作商业盈利目的。
- 经网友介绍，在http://www.pythontik.com找到了由zzj翻译的稿件，经征得同意，转载于此，作为日后学习提高的基础。

内容索引

章节名	翻译者	翻译进度	审校者	审校进度	备注
前言	CliffPeng	100%	CliffPeng	33%	
致谢		0%			
关于本书	CliffPeng	100%	CliffPeng	33%	
第一部分		0%			
第一章 欢迎使用wxPython	ZZJ	100%	CliffPeng	33%	
第二章 给wxPython程序一个坚实的基础	ZZJ	100%	CliffPeng	33%	
第三章 在事件驱动环境中开发	ZZJ	100%	CliffPeng	33%	
第四章 用PyCrust使得					

章节名	翻译者	翻译进度	审校者	审校进度	备注
前言	CliffPeng	100%	CliffPeng	33%	
致谢		0%			
关于本书	CliffPeng	100%	CliffPeng	33%	
第一部分		0%			
第一章 欢迎使用wxPython	ZZJ	100%	CliffPeng	33%	
第二章 给wxPython程序一个坚实的基础	ZZJ	100%	CliffPeng	33%	
第三章 在事件驱动环境中开发	ZZJ	100%	CliffPeng	33%	
第四章 用PyCrust使得wxPython更易处理	ZZJ	100%	CliffPeng	33%	
第五章 绘制蓝图	ZZJ	100%	CliffPeng	33%	
第六章 使用wxPython基本构件	ZZJ	100%	CliffPeng	33%	
第二部分 基础wxPython	ZZJ	100%	CliffPeng	33%	
第七章 使用基础控件	ZZJ	100%	CliffPeng	33%	
第八章 将构件放入窗体中	ZZJ	100%	CliffPeng	33%	
第九章 通过对话框让用户选择	ZZJ	100%	CliffPeng	33%	
第十章 创建和使用wxPython菜单	ZZJ	100%	CliffPeng	33%	
第十一章 使用sizer放置构件	ZZJ	100%	CliffPeng	33%	
第十二章 操作基本图像	ZZJ	100%	CliffPeng	33%	
第三部分 高级wxPython	ZZJ	100%	CliffPeng	33%	
第十三章 建造列表控件并管理列表项	ZZJ	100%	CliffPeng	33%	
第十四章 网格控件	ZZJ	100%	CliffPeng	33%	
第十五章 树形控件	ZZJ	100%	CliffPeng	33%	
第十六章 在应用程序中加入HTML	ZZJ	100%	CliffPeng	33%	
第十七章 wxPython的打印构架	ZZJ	100%	CliffPeng	33%	
第十八章 使用wxPython的其他功能	ZZJ	100%	CliffPeng	33%	

项目日志

- 2007年10月03日，CliffPeng从http://www.pythontik.com处下载了由zjj完成的多数章节翻译稿，并整理了1-12章的部分 `WiKi` 格式。
- 2008年02月23日，CliffPeng在ZQ的开导下重新启动本翻译项目，从zjj的网站下载了更多章节，整理了目录。
- 2008年03月12日，CliffPeng初步完成了辅助（给所有非代码英文增加，😎）脚本编写，进入文本整理阶段。

前言

关于 `Harri Pasanen` 和 `Robin Dunn` 以及 `wxPython` 的传奇故事确切的开始时间是1995年。本书的合著者之一，Robin 写下了下面这段关于 `wxPython` 的文字，而我们决定让故事由参与者自己来讲述，而不是籍由旁人加以引述：

- 1995年，我所进行的一个项目需要在HP-UX系统上部署一个图形用户界面，但我的老板却同时希望在几周后的一个内部展示会上通过Windows3.1掌上电脑来进行一些演示。因此，我开始搜寻跨平台的C++ GUI开发包来进行原型开发。在那个时候，因为没有Google，要完成这样的工作实非易事。但我还是发现了一些商业化的可选开发 包（今天已经无一幸存）和大量的免费开发包。

  当我正在评估分析哪个免费开发包符合当前的紧急需求，并决定哪个商业产品能够满足我们的长期需求时，我偶然看到了 wxWidgets 网站的术语“Python绑定”（这里“绑定”一词指 `Python` 语言和 `wxWidgets` 开发包之间的联系。）由于对如何将软件开发包“绑定”到爬虫动物（在此之前，我从来没有听说过Python语言）充满好奇，我顺着这个链接一个个的点击，直到看到“Python 1.2入门文档”。三个小时后，我从当地的C++倡导者变成了Python的传播者，不断叨扰周边的开发者，向他们展示我的最新发现。

  我开始和芬兰的 `Harri Pasanen` 一起致力于推进 `Python` 和 `wxWidgets` 的绑定（在 `Edward Zimmerman` 的协助下，最终形成了众所周知的 `wxPython 0.2` ），而不是继续我的原型开发。发行声明的邮件列表归档于此处 我们使它的功能强大到多足以让我能够用 `Python` 为老板建造一个原型。但是 `wxPython` 的维护和改进工作却是一个噩梦，因为每件事情（包括 `C++` 扩展模块代码、 `Python` 代理模块、编译系统等等）都是手工完成的， `wxWidget` 的一点点改进都会导致要修改多处 `wxPython` 代码，以确保 `wxPython` 得到改进或修订。当代码增加上几万行时，这种工作方式变得非常笨拙和琐碎。另一个客观事实是：缺乏一个中心源代码库（那是 `SourceForge` 出现之前的时代），因此我们不得不通过电子邮件互相发送代码更新，你可以想象得到这其中的难度。

  大约在此时，我不得不“真正”开始主项目的开发了。在我的支配下，项目开发者们一起进行设计会晤、设定开发期限，从眼中的灵光一现开始，完成整个项目的开发。尽管还可以用 `Python` 来进行一些编译及项目的测试脚本，我发现自己又完全回到了 `C++` 世界。 `Harri` 也无法在上面投入任何时间了，因此 `wxPython` 的开发变得比“爬行”还慢，有时甚至进入了停滞状态。

  1997年，我发现了 `SWIG` （简单封装和接口产生器），并意识到它可以帮助我们解决wxPython项目中困扰已久的所有维护问题。在三、四周的业余时间里，通过使用 `SWIG` ，我几乎将 `wxPython` 的全部重新实现了一遍，而在此前用手工完成这项工作耗用了我几周的全天时间以及Harri几个月的兼职时间。在转向其他项目一段时间后，我发现 `wxWidgets 2.0` 正在积极开发中，而且有了一个全新的架构，因此我不得不重新完成这项工作。但这次，新的架构简

> 化了大量的工作，我仅用了一周的业余时间就完成了！因此，在1998年夏天，第一个“现代版本”的wxPython顺利发布，并从此一直处于活跃开发状态。第一份声明归档于：此处 正如他们所说的，剩下的部分就是传奇故事了。
>
> 必须特别说明的是：`SWIG` 使得我能够轻松创建和维护成千上百行的代码，因此`wxPython`的多种功能特性必须感谢`David Beazley`以及其他项目参与者对`SWIG`的贡献。

通过此书，我们希望能够与您分享对于`wxPython`的热情，在轻松开发图形用户界面应用程序方面，它是真正的独一无二的开发包。我们写作的初衷是不但为初学者，也为专业人员建造一项有用的资源。

致谢

Our work on this book has been aided by a number of people, in ways both obvious and subtle.

Patrick O'Brien was instrumental in getting this project off the ground and Marjan Bace, our publisher at Manning Publications, gave us the chance to make it happen. Jackie Carter was our first editor at Manning and Lianna Wlasiuk came in at a crucial juncture, and, like a great bullpen pitcher, earned the save. We would like to thank our production team at Manning and every-one who worked behind the scenes on the book, especially our copyeditor Elizabeth Martin and our typesetter Denis Dalinnik. Big thanks also to our agent Laura Lewin for her help and understanding.

We've benefited a great deal from the generosity of programmers world-wide in creating the technologies used in this book. First on that list is the crew who developed the wxWidgets foundation of wxPython: Julian Smart, Vadim Zeitlin, Robert Roebling, and the rest of the core wxWidgets team, as well as many contributors. Of course, without Guido van Rossum and other members of the Python language development team, there would be no "Python" in "wxPython." A large part of this book was written using the jEdit text editor and the images were manipulated using the GIMP.

Thanks also to the many reviewers who looked at the manuscript in its vari-ous stages of development and added their valuable insight along the way. They helped make this a better book: Alex Martelli, Dave Brueck, Christopher Bailey, Mike Stok, Jean Baltus, Thomas Palmer, Jack Herrington, Peter Damoc, Nick Coghlan, Pim Van Heuven, John Paulson, Ian Brown, Stefan Neis, Chris Mellon, Kevin Ollivier, and Doug Tillman. Special thanks to Clint Howarth who was our technical proofreader.

NOEL RAPPIN My wife, Erin, makes my life better every day. Thank you for your strength, your intelligence, and your love. I also want to thank my parents Donna and Donnie and my sister Michelle for supporting me fully and com-pletely. Matt Cohen helped this effort by entertaining me via IM while I was try-ing to write. Finally, I want to thank my daughter Emma, who is now old enough to be excited to see her name in a book, and my son Elliot, in anticipation of the day he will also be glad to see his name.

ROBIN DUNN I'd like to thank my wife, Malaura, and my children, Jordyn, Myka, Samuel, and Christian, for their patience, understanding, and love. You make it worth the effort to get past all the difficulties of life. I'd also like to thank the many wonderful members of the wxPython user community whose praise and excitement have helped me to stay motivated and have helped me to take wxPython from an interesting toy to a very useful tool that thousands of develop-ers all around the world love and depend upon. And finally, many thanks go to Mitch Kapor, John Anderson, David Surovell, and others at the Open Source

Applications Foundation for believing in the Open Source concept, the poten-tials of wxPython, and especially for believing in my capabilities and funding my work on wxPython for several years.

关于本书

谁应该阅读本书？

很自然，我们希望每个人都阅读本书。如果你正在书店阅读本书，并在考虑是否购买它，我们会说："买下它！给你们身边的人也买一本，他们今后将会感谢你。"

也就是说，我们撰写本书时，我们头脑中对于读者您做了一定的假设。我们假设您不需要我们解释 `Python` 编程语言的基础。在书中，我们提供了大量的 `Python` 代码。尽管我们认为 `Python` 代码非常好懂，但我们还是认为必须现在就让您知道在书中我们没有提供关于 `Python` 基础的入门指南。如果您需要一份 `Python` 指南，我们向您推荐Manning出版的 `《The Quick Python Book》` ，该书由 `Daryl Harms` 和 `Kenneth McDonald` 编写。

同时我们也假设您至少对描述图形界面对象的基本术语有所熟悉，并从用户角度对图形界面有个大概的了解。至于更加高级的用户界面概念，或者那些不常见的用户显示元素，我们会在用到的时候加以详述。

我们并没有假设您之前就掌握wxPython相关知识。如果您之前就接触过wxPython，我们仍然希望您从本书中能够找到新的信息，或者至少您会发现本书是比现有在线文档更加有用的资源。

本书是如何组织的？

我们将《活学活用wxPython》分成了三个部分。第一部分简要介绍 `wxPython` 的相关概念，并指导读者开始运用wxPython，同时还提供了一些wxPython最佳实践的信息。第一部分的章节包括：

- 第一章 欢迎使用**wxPython** 在该章节中，我们对wxPython进行介绍，并解释为什么说它是自切片面包以来最伟大的事务，同时还提供了用于创建wxPython的一些技术背景资料。

  第二章 给**wxPython**程序一个坚实的基础 讨论了wxPython中最重要的两个对象。每个应用程序都必须有一个应用程序对象和一个顶级窗口。该章将展示如何启动一个wxPython程序，以及如何管理它的生命周期。

第三章 在事件驱动环境中开发 wxPython中的控制是通过事件进行管理的。该章讨论的是如何处理事件（Event），以及如何让它们像钩子一样驱动功能。

第四章 使用 **`PyCrust`** 让**wxPython**更易于处理 `PyCrust` 是用wxPython创建的一种Python外壳，包括了许多高级和有用的特性。你不仅可以使用 `PyCrust` 来进行wxPython开发，你也可以将程序封装于其中来进行调试，而且也可以在自己的应用中对 `PyCrust` 进行复用。

第五章 创建蓝图 该章讨论了在GUI编程者经常遇到困难的领域内的三个最佳实践。我们展示了如何通过重构（ refactoring ）来提升代码的结构和可维护性。通过对 Model/View/Controller 设计模式的探索，我们展示了如何对GUI代码进行单元测试来将错误减少化。

第六章 使用基础构建块 该章是第一部分和第二部分的一座桥梁。在已经展示的基础思想上，我们通过构建一个画板应用对第二、三部分将要讨论的特性进行了一些提示。

第二部分开始了本书更加细节的部分。第二部分中的章节讨论wxPython中最常用的部分内容。这些章节包括了对基础构件集合的入门介绍，讨论了标准窗体和对话框，以及对绘图和布局的相关信息。第二部分中的章节包括：

- 第七章 使用基本控件 该章覆盖了基本构件集合的API函数，包括文本框、按钮、列表框等等。

  第八章 将构件放入窗体 所有的wxPython构件都必须放置在窗体或对话框中。该章覆盖了窗体如何运作、有哪几类窗体、如何管理窗体中的构件。

  第九章 使用对话框给用户选择 对话框的行为方式和窗体的略为不同。我们在该章中讲述了模式对话框以及标准预定以wxPython对话框如何工作。我们同时也展示如何使用wxPython校验器来帮助管理对话框中的数据。

  第十章 创建和使用**wxPython**菜单 多数窗口应用程序具备菜单。我们将展示如何向菜单栏添加菜单，以及如何向菜单中加入菜单项。同时也包含了特殊菜单，比如检查框、无线菜单等内容。我们也将讨论了键盘快捷键和高效使用菜单的有用的指引。

  第十一章 使用尺寸管理器放置构件 在 wxPython 中，尺寸管理器（sizers）被用来减轻手工放置构件这件苦差事带来的痛苦。wxPython中有多个有用的尺寸管理器，我们将向您展示如何使用它们，以及哪种布局最适合哪个管理器。

  第十二章 维护基本图形图像 任何图形界面的基本目标都是在屏幕上画线和形状。在wxPython中，有一系列的可用画图工具可供使用。同时还有一个叫做设备背景（device context）的强大的抽象概念，它使得我们可以向目标作图，而不考虑目标是窗口、打印机还是文件。

第三部分包括了对wxPython更加深入部分的细节讨论。它首先对三个最复杂的wxPython进行了描述，随后讨论了不同的打印和显示机制，最后对一些不值得单独设立章节的有用事务进行了简要介绍。第三部分的章节包括：

- 第十三章 构建列表控制和管理列表项 wxPython的列表控件给了我们以图标模式、列表模式或多列报告模式显示“浏览风格”列表的能力。你也可以定义排序行为，并允许用户对列表条目进行编辑。

  第十四章，调整网格控件 如果你需要像电子表格一样的东西，wxPython的网格控件是符合你需求的完整功能构件。它允许对网格显示和行为的彻底控制，并允许完全的自定义配置。

  第十五章 “爬上”树型控件 wxPython 树型控件使得我们可以对分等级的数据进行紧凑显示，包括但不限于目录树或者类的继承关系。你也可以允许用户任意对条目进行编辑。

  第十六章 向**wxPython**应用程序中并入**HTML** 在wxPython中，你可以使用HTML来简化格式化文本的显示和打印。wxPython中的HTML引擎可以定义以用于满足特殊需求。

  第十七章 **wxPython**打印框架体系 wxPython中的打印通过一些专门的打印、打印数据和打印预览对象来进行管理。在该章中，我们对它们如何一起工作进行了探索。

  第十八章 使用其他**wxPython**功能 在该章中，我们讲述了一些重要的特性，它们还没有长到可以单独设立章列的地步，包括剪切、粘贴、拖放和多线程。

如何使用本书

如何使用本书取决于您对wxPython的掌握了解程度我们对本书的策划目标是要对专家和初学者都有所帮助，但也希望不同层次的用户对本书的不同部分都有或多或少有所共鸣。

如果你是wxPython初学者，你一定会想从第一部分开始。第一章至第三章帮助您夯实wxPython相关概念的坚实基础。第六章则对构建合理大小程序的步骤进行了完整回顾。第五章介绍如何让代码更易于管理的方法，第四章提供了一些协助进行调试、编写wxPython应用的工具。当你开始编写自己的wxPython程序时，你就已经开始使用第二部分讨论的API了——我们试图按照功能进行组织章节内容，以方便您查找有用的主题。

如果您对wxPython已经熟悉了，你可能将时间主要花在第二部分和第三部分之上。然而，我们依然推荐您快速阅读一遍第一部分。如果您不熟悉 `PyCrust` ，第四章对您来说就是新内容了，我们相信在第五章中您也能找到一些有用的内容。

在第三部分，您将会发现一些更加复杂的构件的讨论，您将发现这些章节中的代码比其他章节中的更长也更加完整。 本书中的这些例子都是基于Python 2.3.x编写的——我想我们没有包括所有Python 2.4和wxPython 2.5中的新特性。wxPython 2.6.x版本发布比较迟了，因此本书无法涵盖其内容。然而，它确实包含了大量的BUG补丁和对wxWidgets的有限兼容。

正式开始之前，还有一点我们必须指明。那就是本书并不准备事无巨细地提供wxPython的所有内容的参考。我们期望本书能够为您需要知道的所有功能特性提供参考，但它确实没有100%覆盖所有的特性。出于时间和空间的考虑，我们不得不选择将重点放在某些元素上，而其他的一些则无法详细介绍。例如，有一些从C++

wxWidgets中继承的wxPython特性在标准Python库中得到了复制，对于这些特性我们选择不在本书中涵盖。同时，如果你使用了1990年代的Windows操作系统，您可能会发现一些例子中的特性并非如所描述的那样起工作，在此我们没有足够的空间枚举出这些例外。最后，有一些核心构件集合的特性我们认为并不经常用到，因此我们也没有空间公平对待它们。

印刷约定

本书中始终使用一下约定：

- Courier 字体使用在所有代码列表中
- Italics 字体用来介绍新术语
- Courier Bold 字体有时用来吸引您对部分代码的注意。
- 代码注释用来将您的注意力指向特定的代码行注释采用句点标记，比如 b.
- Courier 字体用于代码中的文本，wxPython类和方法名称，或者Python代码片段。

代码下载

本书中使用的所有例子的源代码可以从出版商的网站 http://www.manning.com/rappin下载。

从何处获得更多帮助

虽然我们尽可能地让本书易于理解，但我们不可能预见您你在使用wxPython过程中可能预见的所有用法和问题。

wxPython的主站http://www.wxpython.org有一些资源也许可以帮助您解决问题。

官方在线文档位置为：wxpython.org/docs/api/

在http://wiki.wxpy-thon.org/有一个互助wiki网站，当然你也可以订阅http://www.wxpy-thon.org/maillist.php处的邮件列表。

作者在线

也可以从作者在线论坛获取帮助，这是一个由Manning出版商提供的私人网页论坛。您可以通过该论坛对本书进行评论，咨询技术问题，接受来自作者或其他读者的帮助。使用浏览器访问http://www.man-ning.com/rappin可获取这些免费服务。论坛的欢迎页面给出了注册和其他行为所需的所有信息。 作者在线论坛（ Author Online forum）是Manning始终忠于读者的表现方式之一。作者对论坛中的参与是完全资源的，没有任何特定级别的义务性。论坛是与他人分享思路和向他人学习的好途径。主要本书仍在销售中，就可以出版商的网站进入作者在线论坛。

关于书名

通过组合介绍、回顾和“如何”例子，“活学活用”书籍设计意图是帮助学习和记忆。按照认知科学的研究成果，人们记住的事物是他们在自主探索中获得的事物。 尽管 `Manning` 公司没有谁是认知科学家，但我们确信要想让所学变成永久财富，必须经历探索、实际动手操作，（有趣地是还包括）对所学内容的重述等步骤。人们对新事务进行理解和记忆，也就是说只有在积极探索之后，人们才会掌握它们。人们总是在使用的过程中学习。“活学活用”引导的基本部分是实例驱动的。它鼓励读者自己尝试，对新代码实际操作以及探索新的思路。 对于本书的书名来说，还有一条更加通俗的理由，我们的读者都很忙碌。他们使用书籍往往是为了完成某项工作或解决某个问题。他们需要的书籍能够允许他们自由跳跃式阅读，只在需要某个知识点的时候才对它进行学习。他们需要书籍使自己处于“运转”状态。该系列的书籍是为这样的读者设计的。

关于封面绘图

《活学活用wxPython》一书的封面图形为“Soldat Japonais”，一名日本士兵。该插图取自1796年法国出版的由 `J. G. St. Saveur` 所著的法文旅游书籍 `《Encyclopedie des Voyages》` 。在那个时候，为寻求快乐而旅行还是个相对新鲜的现象，像该书一样的旅游指南非常受欢迎，它不仅介绍了旅程者自己，还反映了惬意的旅行者前往法国其他地区及国外居住地的见闻。

`《Encyclopedie des Voyages》` 一书中多样的插图生动地反映了200多年前世界各地的城镇和省份的独特和个性。这个时代，两个相隔仅数英里的地区会通过服饰代码来区分人所属地。这本旅行指南在狂热的现实中唤醒了我们关于那个时代及其他历史时期的那种与世隔绝的感受。

那个年代之后，服饰代码已经发生变化，地区差异也随时代变迁逐渐褪色。现在，我们经常会很难区分居住不同大洲的居民。也许，乐观点来说，我们已经将文化、可见差异与更加多样性的个体生活做了一次交易。或者说一个更多样的充满趣味的的理性、技术性生活。 在 `Manning` ，我们通过该旅行指南中的图片将两个世纪之前地区生活间的丰富差异带回现实，以此赞美计算机商业的独创性、能动性与乐趣性。

第一部分

第一章 欢迎使用wxPython

下面是一个例子，它创建了一个有一个文本框的窗口用来显示鼠标的位置。

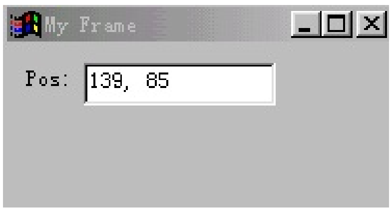
```
#!/bin/env python
import wx
class MyFrame(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, -1, "My Frame", size=(300,
 300))
        panel = wx.Panel(self, -1)
        panel.Bind(wx.EVT_MOTION,  self.OnMove)
        wx.StaticText(panel, -1, "Pos:", pos=(10, 12))
        self.posCtrl = wx.TextCtrl(panel, -1, "", pos=(40, 10))

    def OnMove(self, event):
        pos = event.GetPosition()
        self.posCtrl.SetValue("%s, %s" % (pos.x, pos.y))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show(True)
    app.MainLoop()
```

图示如下：



漂亮的界面是一个 `GUI` 程序必不可少的一部分， `wxPython` 可以做到这一点，加之 `Python` 强大的功能和简洁的语法，使用得它在 `Python` 的 `gui` 中成为一种主流。

开始wxPython

首先我们创建一个显示一个图像的文件。这将分三步：

1. 首先创建一个空的最小的可以工作的 `wxPthon` 程序
2. 组织和细化
3. 显示 `wxPython` 的 `logo`

图示如下：



Figure 1.2
Running hello.py
on Windows

创建最小的空的wxPython程序

我们创建一个名为 `bare.py` 的程序并键入以下代码：

```
import wx #1

class App(wx.App):#2

    def OnInit(self): #3
        frame = wx.Frame(parent=None, title='Bare')
        frame.Show()
        return True

app = App() #4
app.MainLoop() #5
```

上面的代码运行的结果如下：



Figure 1.5
Running bare.py on Windows.

上面的代码的任何一行都不能少，否则将不能工作。这个基本的 `wxPython` 程序说明了开发任一 `wxPython` 程序所必须的五个基本步骤：

1. 导入必须的 `wxPython` 包
2. 子类化 `wxPython` 应用程序类
3. 定义一个应用程序的初始化方法
4. 创建一个应用程序类的实例
5. 进入这个应用程序的主事件循环

下面让我们看看这个最小的空的程序是如何一步一步实现的。

导入wxPython

你需要做的第一件事就是导入这个主要的 `wxPython` 包，这个包名为 `wx:`

```
import wx
```

一旦这个包被导入，你就可以引用 `wxPython` 的类、函数和常量（它们以 `wx` 为前缀），如下所示：

```
class App(wx.App):
```

注意：老的引入方式仍然被支持，你可能会遇到用这种老的引入方式的代码。因此我们将会简短地说明这种老的方式及为什么要改变它。老的包的名字是 `wxPython` ，它包含了一个内在的名为 `wx` 模块。那时，通常有两种导入必要的代码的方法，一种就是从 `wxPython` 包中导入 `wx` 模块： `from` `wxPython` `import` `wx` ；另一种就是直接从 `wx` 模块中导入所有的东西： `from` `wxPython.wx` `import` 。这两种方法都有严重的缺点。这第二种方法 *`Python`* 中是不建议使用的，这因为可能导致名字空间冲突，而老的 *`wx`* 模块通过在其属性前加一个 *`wx`* 前缀避免了这个问题。尽管使用这个安全防范，但是 *`import`* 仍然有可能导致问题，但是许多 `wxPython` 程序员喜欢这种类型，并且你将在老的代码中经常看到这种用法。这种风格的坏处是类名以小写字母开头，而大多数 `wxPython` 方法以大写字母开头，这和通常的 `Python` 编写程序的习惯相反。

然而如果你试图避免由于使用 `import` *导致的名字空间膨胀，而使用 `from` `wxPython` `import` `wx` 。那么你就不得不为每个类、函数、常数名键入两次 `wx` ，一次是作为包的前缀，另一次是作为通常的前缀，例如 `wx.wxWindow` 。

对于导入顺序需要注意的是：你从 `wxPython` 导入其它东西之前必须先导入 `wx` 。通常情况下， `Python` 中的模块导入顺序无关。但是 `wxPython` 中的不同，它是一个复杂的模块。当你第一次导入 `wx` 模块时， `wxPython` 要对别

的 `wxPython` 模块执行一些初始化工作。例如 `wxPython` 中的一些子包，如 `xrc` 模块，它在 `wx` 模块导入之前不能够正确的工作，我们必须按下面顺序导入：

```
import wx
from wx import xrc
```

以上的导入顺序只针对 `wxPython` 的模块， `Python` 的模块导入顺序没关系。例如：

```
import sys
import wx
import os
from wx import xrc
import urllib
```

使用应用程序和框架工作

一旦你导入了 `wx` 模块，你就能够创建你的应用程序（ `application` ）对象和框架（ `frame` ）对象。每个 `wxPython` 程序必须有一个 `application` 对象和至少一个 `frame` 对象。 `application` 对象必须是 `wx.App` 的一个实例或你在 `OnInit()` 方法中定义的一个子类的一个实例。当你的应用程序启动的时候， `OnInit()` 方法将被 `wx.App` 父类调用。

子类化 **`wxPython`** **`application`** 类

下面的代码演示了如何定义我们的 `wx.App` 的子类：

```
class MyApp(wx.App):

    def OnInit(self):
        frame = wx.Frame(parent=None, id=-1, title="Bare")
        frame.Show()
        return True
```

上面我们定义了一个名为 `MyApp` 的子类。我们通常在 `OnInit()` 方法中创建 `frame` 对象。上面的 `wx.Frame` 接受三个参数，仅第一个是必须的，其余的都有默认值。 调用 `Show()` 方法使 `frame` 可见，否则不可见。我们可以通过给 `Show()` 一个布尔值参数来设定 `frame` 的可见性：

```
frame.Show(False)  # 使框架不可见.
frame.Show(True)   # True是默认值，使框架可见.
frame.Hide()       # 等同于frame.Show(False)
```

定义一个应用程序的初始化方法

注意：我们没有为我们的应用程序类定义一个 `__init__()` 方法。
在 `Python` 中，这就意味着父方法 `wx.App.__init()__` 将在对象创建时被自动调用。这是一个好的事情。如果你定义你自己的 `__init__()` 方法，不要忘了调用其基类的 `__init()__` 方法，示例如下：

```
class App(wx.App):
    def __init__(self):

        wx.App.__init__(self)
```

如果你忘了这样做， `wxPython` 将不被初始化并且你的 `OnInit()` 方法也将得不到调用。

创建一个应用程序实例并进入它的主事件循环

这步是创建 `wx.App` 子类的实例，并调用它的 `MainLoop()` 方法：

```
app = App()
app.MainLoop()
```

一旦进入主事件循环，控制权将转交给 `wxPython` 。 `wxPython` `GUI` 程序主要响应用户的鼠标和键盘事件。当一个应用程序的所有框架被关闭后，这个 `app.MainLoop()` 方法将返回且程序退出。

扩展这个最小的空的wxPython程序

现在我们将给空的最小程序增加适当数量的功能，它包含了通常 `Python` 编程的标准并能够作为你自己的程

序的一个基准。下面我们创建一个名为 `spare.py` 程序：

```
#!/usr/bin/env python   #1

"""Spare.py is a starting point for a wxPython program."""   #2

import wx

class Frame(wx.Frame):   #3
    pass

class App(wx.App):

    def OnInit(self):
        self.frame = Frame(parent=None, title='Spare')   #4
        self.frame.Show()
        self.SetTopWindow(self.frame)   #5
        return True

if __name__ == '__main__':   #6
    app = App()
    app.MainLoop()
```

这个程序仍然很小，只有14行代码，但是它增加了几个重要的项目让我们考虑到什么样的代码是好的、完整的。

#1 这行看似注释，但是在如 `linux` 和 `unix` 等操作系统上，它告诉操作系统如何找到执行该程序的解释器。如果这个程序被给予可执行权限（例如使用 `chmod` 命令），我们可以在命令行下仅仅键入该程序的名字来运行这个程序：

```
% spare.py
```

这行在其它的操作系统上将被忽略。但是包含它可以实现代码的跨平台。

#2 这是文档字符串，当模块中的第一句是字符串的时候，这个字符串就成了该模块的文档字符串并存储

在该模块的 `__doc__` 属性中。你能够在你的代码中、某些开发平台、甚至交互模式下运行的 `Python` 解释器

中访问文档字符串：

```
    import spare
    print spare.__doc__
```

`Spare.py` 简单 `wxPython` 程序的起点。

#3 我们改变了你们创建 `frame` 对象的方法。 `bare` 版的程序简单地创建了一个 `wx.Frame` 类的实例。在 `spare` 版中，我们定义了我们自己的 `Frame` 类作为 `wx.Frame` 的子类。此时，最终的结果没有什么不同，但是如果你想在你的框架中显示诸如文本、按钮、菜单的话，你可能就想要你自己的 `Frame` 类了。

#4 我们将对 `frame` 实例的引用作为应用程序实例的一个属性

#5 在 `OnInit()` 方法中，我们调用了这个 `App` 类自己的 `SetTopWindow()` 方法，并传递给它我们新创建的 `frame` 实例。我们不必定义 `SetTopWindow()` 方法，因为它继承自 `wx.App` 父类。 `SetTopWindow()` 方法是一个可选的方法，它让 `wxPython` 方法知道哪个框架或对话框将被认为是主要的。一个 `wxPython` 程序可以有几个框架，其中有一个是被设计为应用程序的顶级窗口的。

#6 这个是 `Python` 中通常用来测试该模块是作为程序独立运行还是被另一模块所导入。我们通过检查该模块的 `__name__` 属性来实现：

```
if __name__ == '__main__':
    app = App()
    app.MainLoop()
```

创建最终的hello.py程序

代码如下：

```
#!/usr/bin/env python

"""Hello, wxPython! program."""

import wx

class Frame(wx.Frame):   #2 wx.Frame子类
    """Frame class that displays an image."""

    def __init__(self, image, parent=None, id=-1,
                 pos=wx.DefaultPosition,
                 title='Hello, wxPython!'): #3图像参数
        """Create a Frame instance and display image."""
#4 显示图像
        temp = image.ConvertToBitmap()
        size = temp.GetWidth(), temp.GetHeight()
        wx.Frame.__init__(self, parent, id, title, pos, size)
        self.bmp = wx.StaticBitmap(parent=self, bitmap=temp)

class App(wx.App):  #5 wx.App子类
    """Application class."""

    def OnInit(self):
#6 图像处理
        image = wx.Image('wxPython.jpg', wx.BITMAP_TYPE_JPEG)
        self.frame = Frame(image)

        self.frame.Show()
        self.SetTopWindow(self.frame)
        return True

def main():  #7
    app = App()
    app.MainLoop()

if __name__ == '__main__':
     main()
```

说明：

- **#2** 定义一个 `wx.Frame` 的子类，以便我们更容量控制框架的内容和外观。

  #3 给我们的框架的构造器增加一个图像参数。这个值通过我们的应用程序类在创建一个框架的实例时提供。同样，我们可以传递必要的值给 `wx.Frame.__init__()`

  #4 我们将用 `wx.StaticBitmap` 控件来显示这个图像，它要求一个位图。所以我们转换图像到位图。我们也使用图像的宽度和高度创建一个 `size` 元组。这个 `size` 元组被提供给 `wx.Frame.__init__()` 调用，以便于框架的尺寸匹配位图尺寸。

#5 定义一个带有 `OnInit()` 方法的 `wx.App` 的子类，这是 `wxPython` 应用程序最基本的要求。

#6 我们使用与 `hello.py` 在同一目录下的名为 `wxPython.jpg` 的文件创建了一个图像对象。

#7 `main()` 函数创建一个应用程序的实例并启动 `wxPython` 的事件循环。

第二章 给wxPython程序一个坚实的基础

房屋的基础是混凝土结构，它为其余的建造提供了坚固的基础。你的 `wxPython` 程序同样有一个基础，它由两个必要的对象组成，用于支持你的应用程序的其余部分。它们是应用程序对象和顶级窗口对象。适当地使用这两个对象将给你的 `wxPython` 应用程序一个稳固的开始并使得构造你的应用程序的其余部分更容易。

关于所要求的对象我们需要知道些什么？

让我们来说明一下这两个基础对象。应用程序对象管理主事件循环，主事件循环是你的 `wxPython` 程序的动力。启动主事件循环是应用程序对象的工作。没有应用程序对象，你的 `wxPython` 应用程序将不能运行。 顶级窗口通常管理最重要的数据，控制并呈现给用户。例如，在词处理程序中，主窗口是文档的显示部分，并很可能管理着该文档的一些数据。类似地，你的 `web` 浏览器的主窗口同时显示你所关注的页面并把该页作为一个数据对象管理。 下图显示了这两个基础对象和你的应用程序的其它部分这间的关系：

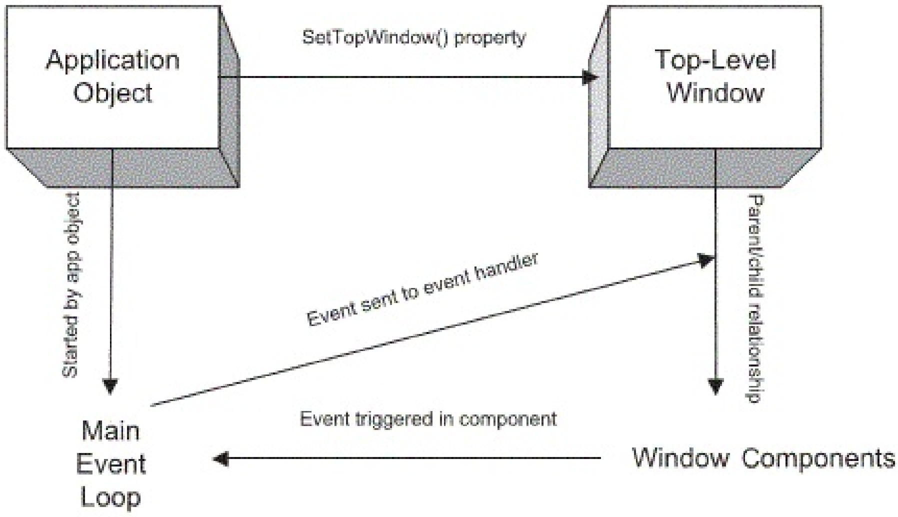


Figure 2.1 A schematic of the basic wxPython application structure, showing the relationship between the application object, the top-level window, and the main event loop

如图所示，这个应用程序对象拥有顶级窗口和主事件循环。顶级窗口管理其窗口中的组件和其它的你分配给它的数据对象。窗口和它的组件的触发事件基于用户的动作，并接受事件通知以便改变显示。

如何创建和使用一个应用程序对象？

任何 `wxPython` 应用程序都需要一个应用程序对象。这个应用程序对象必须是类 `wx.App` 或其定制的子类的一个实例。应用程序对象的主要目的是管理幕后的主事件循环。这个事件循环响应于窗口系统事件并分配它们给适当的事件处理器。这个应用程序对象对 `wxPython` 进程的管理如此的重要以至于在你的程序没有实例化一个应用程序对象之前你不能创建任何的 `wxPython` 图形对象。

父类 `wx.App` 也定义了一些属性，它们对整个应用程序是全局性的。很多时候，它们就是你对你的应用程序对象所需要的全部东西。假如你需要去管理另外的全局数据或连接（如一个数据库连接），你可以定制应用程序子类。在某些情况下，你可能想为专门的错误或事件处理而扩展这个主事件循环。然而，默认的事件循环几乎适合所有的你所要写的 `wxPython` 应用程序。

创建一个**wx.App**的子类

创建你自己的 `wx.App` 的子类是很简单的。当你开始你的应用程序的时候，创建你自己的 `wx.App` 的子类通常是一个好的想法，即使是你不定制任何功能。创建和使用一个 `wx.App` 子类，你需要执行四个步骤：

1、定义这个子类 2、在定义的子类中写一个 `OnInit()` 方法 3、在你的程序的主要部分创建这个类的一个实例 4、调用应用程序实例的 `MainLoop()` 方法。这个方法将程序的控制权转交给 `wxPython`

我们在第一章中看到过 `OnInit()` 方法。它在应用程序开始时并在主事件循环开始前被 `wxPython` 系统调用。这个方法不要求参数并返回一个布尔值，如果所返回的值是 `False` ，则应用程序将立即退出。大多数情况下，你将想要该方法返回的结果为真。处理某些错误条件，退出可能是恰当的方法，诸如所一个所需的资源缺失。

由于 `OnInit()` 方法的存在，并且它是 `wxPython` 架构的一部分，所以任何关于你的定制的类的所需的初始化通常都由 `OnInit()` 方法管理，而不在 `Python` 的 `__init__` 方法中。如果由于某些原因你决定需要 `__init__` 方法，那么你必须在你的 `__init__` 方法中调用父类的 `__init__` 方法，如下所示：

```
wx.App.__init__(self)
```

通常，你在 `OnInit()` 方法中将至少创建一个框架对象，并调用该框架的 `Show()` 方法。你也可以有选择地通过调用 `SetTopWindow()` 方法来为应用程序指定一个框架作为顶级窗口。顶级窗口被作为那些没有指定父窗口的对话框的默认父窗口。

何时省略 **`wx.App`** 的子类

你没有必要创建你自己的 `wx.App` 子类，你通常想这样做是为了能够在 `OnInit()` 方法中创建你的顶级框架。 通常，如果在系统中只有一个框架的话，避免创建一个 `wx.App` 子类是一个好的主意。在这种情况下， `wxPython` 提供了一个方便的类 `wx.PySimpleApp` 。这个类提供了一个最基本的 `OnInit()` 方法， `wx.PySimpleApp` 类定义如下：

```
class PySimpleApp(wx.App):

    def __init__(self, redirect=False, filename=None,
                 useBestVisual=False, clearSigInt=True):
        wx.App.__init__(self, redirect, filename, useBestVisual,
                clearSigInt)

    def OnInit(self):
        return True
```

下面是 `wx.PySimpleApp` 一个简单用法：

```
if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyNewFrame(None)
    frame.Show(True)
    app.MainLoop()
```

在上面这段代码的第一行，你创建了一个作为 `wx.PySimpleApp` 的实例的应用程序对象。由于我们在使用 `wx.PySimpleApp` 类，所以我们没有定制 `OnInit` 方法。第二行我们定义了一个没有父亲的框架，它是一个顶级的框架。（很显然，这个 `MyNewFrame` 类需要在别处被定义）这第三行显示框架，最后一行调用应用程序主循环。

正如你所看到的，使用 `wx.PySimpleApp` 让你能够运行你的 `wxPython` 程序而无需创建你自己定制的应用程序类。如果你的应用程序十分简单的话，你应该只使用 `wx.PySimpleApp` ，且不需要任何其它的全局参数。

理解应用程序对象的生命周期

你的 `wxPython` 应用程序对象的生命周期开始于应用程序实例被创建时，在最后一个应用程序窗口被关闭时结束。这个没有必要与你的 `wxPython` 应用程序所在的 `Python` 脚本的开始和结束相对应。 `Python` 脚本可以在 `wxPython` 应用程序创建之前选择做一动作，并可以在 `wxPython` 应用程序的 `MainLoop()` 退出后做一些清理工作。然而所有的 `wxPython` 动作必须在应用程序对象的生命周期中执行。正如我们曾提到过的，这意味你的主框架对象在 `wx.App` 对象被创建之前不能被创建。（这就是为什么我们建议在 `OnInit()` 方法中创建顶级框架——因为这样一来，就确保了这个应用程序已经存在。）

下图所示，创建应用程序对象触发 `OnInit()` 方法并允许新的窗口对象被创建。在 `OnInit()` 之后，这个脚本调用 `MainLoop()` 方法，通知 `wxPython` 事件现在正在被处理。在窗口被关闭之前应用程序继续它的事件处理。当所有顶级窗口被关闭后， `MainLoop()` 函数返回同时应用程序对象被注销。这之后，这个脚本能够关闭其它的可能存丰的连接或线程。

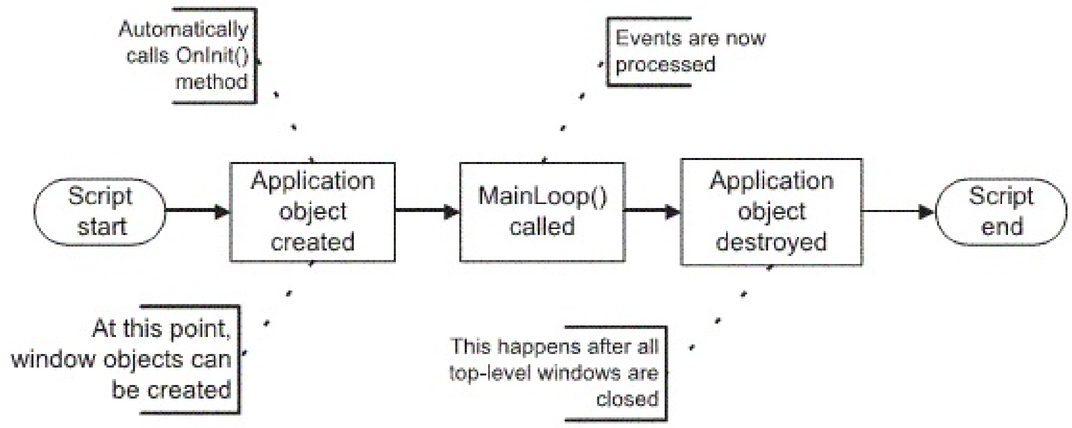


Figure 2.2 Major events in the wxPython application lifecycle, including the beginning and ending of both the wxPython application and the script which surrounds it

如何定向wxPython程序的输出？

所有的 `Python` 程序都能够通过两种标准流来输出文本：分别是标准输出流 `sys.stdout` 和标准错误流 `sys.stderr` 。通常， `Python` 脚本定向标准输出流到它所运行的控制台。然而，当你的应用程序对象被创建时，你可以决定使

用 `wxPython` 控制标准流并重定向输出到一个窗口。在 `Windows` 下，这个重定向行为是 `wxPython` 的默认行为。而在 `Unix` 系统中，默认情况下， `wxPython` 不控制这个标准流。在所有的系统中，当应用程序对象被创建的时候，重定向行为可以被明确地指定。我们推荐利用这个特性并总是指定重定向行为来避免不同平台上的不同行为产生的任何问题。

重定向输出

如果 `wxPython` 控制了标准流，那么经由任何方法发送到流的文本被重定向到一个 `wxPython` 的框架。在 `wxPyton` 应用程序开始之前或结束之后发送到流的文本将按照 `Python` 通常的方法处理（输出到控制台）。下例同时演示了应用程序的生命周期和 `stdout` / `stderr` 重定向：

```
#!/usr/bin/env python

import wx
import sys

class Frame(wx.Frame):

    def __init__(self, parent, id, title):
        print "Frame __init__"
        wx.Frame.__init__(self, parent, id, title)

class App(wx.App):

    def __init__(self, redirect=True, filename=None):
        print "App __init__"
        wx.App.__init__(self, redirect, filename)

    def OnInit(self):
        print "OnInit"    #输出到stdout
        self.frame = Frame(parent=None, id=-1, title='Startup')
 #创建框架
        self.frame.Show()
        self.SetTopWindow(self.frame)
        print    sys.stderr, "A pretend error message"    #输出到
stderr
        return True

    def OnExit(self):
        print "OnExit"

if __name__ == '__main__':
    app = App(redirect=True) #1 文本重定向从这开始
    print "before MainLoop"
    app.MainLoop()  #2 进入主事件循环
    print "after MainLoop"
```

说明：

#1 这行创建了应用程序对象。这行之后，所有发送到 `stderr` 或 `stdout` 的文本都可被 `wxPython` 重定向到一个框架。参数 `redirect = True` 决定了是否重定向。

#2 运行的时候，应用程序创建了一个空的框架和也生成了一个用于重定向输出的框架。图示如下：

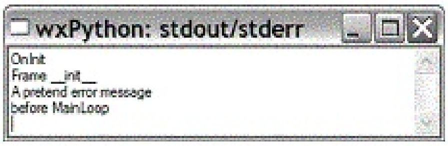


Figure 2.3 The stdout/stderr window created by Listing 2.1

注意： `stdout` 和 `stderr` 都定向到这个窗口。 当你运行了这个程序之后，你将会看到你的控制台有下面的输出：

`App __init__ after MainLoop`

这第一行在框架被打开之前生成，第二行在框架被关闭之后生成。 通过观察控制台和框架的输出，我们可以跟踪应用程序的生命周期。

下面我们将上面的程序与图2.2作个比较，图中的" `Start Script` "对应于程序的 `__main__` 语句。然后立即过渡到下一" `Application obect created` ",对应于程序的 `app = App(redirect = True)` 。应用程序实例的创建通过调用 `wx.App.__init__()` 方法。然后是 `OnInit()` ，它被 `wxPython` 自动调用。从这里，程序跳转到 `wx.Frame.__init__()` ，它是在 `wx.Frame` 被实例化时运行。最后控制转回到 `__main__` 语句，这里， `MainLoop()` 被调用，对应于图中的" `MainLoop() called` "。主循环结束后， `wx.App.OnExit()` 被 `wxPython` 调用，对应于图中" `Application object destroyed` "。然后脚本的其余部分完成处理。

为什么来自 `OnExit()` 的消息既没显示在窗口中也没显示在控制台中呢？其实它是在窗口关闭之前显示在 `wxPython` 的框架中，但窗口消失太快，所以无法被屏幕捕获。

修改默认的重定向行为

为了修改这个行为， `wxPython` 允许你在创建应用程序时设置两个参数。第一个参数是 `redirect` ，如果值为 `True` ，则重定向到框架，如果值为 `False` ，则输出到控制台。如果参数 `redirect` 为 `True` ，那么第二个参数 `filename` 也能够被设置，这样的话，输出被重定向到 `filename` 所指定的文件中而不定向到 `wxPython` 框架。因此，如果我们将上例中的 `app = App(redirect = True)` 改为 `app = App(False)` ，则输出将全部到控制台中：

```
App __init__
OnInit
Frame __init__
A pretend error message
before MainLoop
OnExit
after MainLoop
```

我们可以注意到`OnExit()`消息在这里显示出来了。我们再作一个改变：

```
app = App(True, "output")
```

这将导致所有的应用程序创建后的输出重定向到名为`output`的文件中。而"`App__init`"和"`after` `MainLoop`"消息仍将发送到控制台，这是因为它们产生在`wx.App`对象控制流的时期之外。

如何关闭wxPython应用程序?

当你的应用程序的最后的顶级窗口被用户关闭时，`wxPython`应用程序就退出了。我们这里所说的顶层窗口是指任何没有父亲的框架，并不只是使用`SetTopWindow()`方法设计的框架。这包括任何由`wxPython`自身创建的框架。在我们重定向的例子中，`wxPython`应用程序在主框架和输出重定向的框架都被关闭后退出，仅管只有主框架是使用`SetTopWindow()`登记的，尽管应用程序没有明确地创建这个输出重定向框架。要使用编程触发一个关闭，你可以在所有的这里所谓顶级窗口上调用`Close()`方法。

管理正常的关闭

在关闭的过程期间，`wxPython`关心的是删除所有的它的窗口和释放它们的资源。你可以在退出过程中定义一个钩子来执行你自己的清理工作。由于你的`wx.App`子类的`OnExit()`方法在最后一个窗口被关闭后且在`wxPython`的内在的清理过程之前被调用，你可以使用`OnExit()`方法来清理你创建的任何非`wxPython`资源（例如一个数据库连接）。即使使用了`wx.Exit()`来关闭`wxPython`程序，`OnExit()`方法仍将被触发。

如果由于某种原因你想在最后的窗口被关闭后`wxPython`应用程序仍然可以继续，你可以使用`wx.App`的`SetExitOnFrameDelete(flag)`方法来改变默认的行为。如果`flag`参数设置为`False`，那么最后的窗口被关闭后`wxPython`应用程序仍然会继续运行。这意味着`wx.App`实例将继续存活，并且事件循环将继续处理事件，比如这时你还可以创建所有新的这里所谓的顶级窗口。`wxPython`应用程序将保持存活直到全局函数`wx.Exit()`被明确地调用。

管理紧急关闭

你不能总是以一个可控的方法关闭你的程序。有时候，你需要立即结束应用程序并不考虑清理工作。例如一个必不可少的资源可能已被关闭或被损坏。如果系统正在关闭，你可能不能做所有的清理工作。

这里有两种在紧急情况下退出你的 `wxPython` 应用程序的方法。你可以调用 `wx.App` 的 `ExitMainLoop()` 方法。这个方法显式地使用主消息循环终止，使用控制离开 `MainLoop()` 函数。这通常将终止应用程序，这个方法实际上等同于关闭所有这里所谓顶级窗口。

你也可以调用全局方法 `wx.Exit()` 。正常使用情况下，两种方法我们都不推荐，因为它将导致一些清理函数被跳过。

有时候，你的应用程序由于一个控制之外的事件而需要关闭。例如操作系统的关闭或注销。在这种情况下，你的应用程序将试图做一些保存文档或关闭连接等等。如果你的应用程序为 `wx.EVT_QUERY_END_SESSION` 事件绑定了一个事件处理器，那么当 `wxPython` 得到关闭通知时这个事件处理器将被调用。这个 `event` 参数是 `wx.CloseEvent` 。我们可以通过关闭事件来否决关闭。这可以使用关闭事件的 `CanVeto()` 方法， `CanVeto()` 方法决定是否可以否决， `Veto()` 执行否决。如果你不能成功地保存或关闭所有的资源，你可能想使用该方法。 `wx.EVT_QUERY_END_SESSION` 事件的默认处理器调用顶级窗口的 `Close()` 方法，这将依次向顶层窗口发送 `wx.EVT_CLOSE` 事件，这给了你控制关闭过程的另一选择。如果任何一个 `Close()` 方法返回 `False` ，那么应用程序将试图否决关闭。

如何创建和使用顶级窗口对象?

在你的应用程序中一个顶级窗口对象是一个窗口部件（通常是一个框架），它不被别的窗口部件所包含。顶级窗口对象通常是你的应用程序的主窗口，它包含用户与之交互的窗口部件和界面对象。当所有的顶级窗口被关闭时应用程序退出。

你的应用程序至少必须有一个顶级窗口对象。顶级窗口对象通常是类 `wx.Frame` 的子类，尽管它也可以是 `wx.Dialog` 的子类。大多数情况下，你将为了使用为你的应用程序定义定制的 `wx.Frame` 的子类。然而，这儿也存在一定数量的预定义的 `wx.Dialog` 的子类，它们提供了许多你可能会在一个应用程序中遇到的典型的对话框。

这儿可能有一个名称上的混淆，那就是“顶级窗口”。一般意义上的顶级窗口是指在你的应用程序中任何没有父容器的窗口部件。你的应用程序必须至少有一个，但是，只要你喜欢可以有多个。但是它们中只有一个可以通过使用 `SetTopWindow()` 被 `wxPython` 作为主顶级窗口。如果你没有使用 `SetTopWindow()` 指定主顶级窗口，那么在 `wx.App` 的顶级窗口列表中的第一个框架将被认为是这个主顶级窗口。因此，明确地定义一个主顶级窗口不总是必要的，例如，你只有一个顶级窗口的时候。反复调用 `SetTopWindow()` 将反复改变当前的主顶级窗口，因为一个应用程序一次只能有一主顶级窗口。

使用wx.Frame

按照 `wxPython` 中的说法，框架就是用户通常称的窗口。那就是说，框架是一个容器，用户可以将它在屏幕上任意移动，并可将它缩放，它通常包含诸如标题栏、菜单等等。在 `wxPython` 中， `wx.Frame` 是所有框架的父类。这里也有少数专用的 `wx.Frame` 子类，你可以使用它们。

当你创建 `wx.Frame` 的子类时，你的类应该调用其父类的构造器 `wx.Frame.__init__()` 。 `wx.Frame` 的构造器所要求的参数如下：

```
wx.Frame(parent, id=-1, title="", pos=wx.DefaultPosition,
        size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE,
        name="frame")
```

我们在别的窗口部件的构造器中将会看到类似的参数。参数的说明如下：

`parent` ：框架的父窗口。对于顶级窗口，这个值是 `None` 。框架随其父窗口的销毁而销毁。取决于平台，框架可被限制只出现在父窗口的顶部。在多文档界面的情况下，子窗口被限制为只能在父窗口中移动和缩放。

`id` ：关于新窗口的 `wxPython` `ID` 号。你可以明确地传递一个。或传递-1，这将导致 `wxPython` 自动生成一个新的 `ID` 。

`title` ：窗口的标题。

`pos` ：一个 `wx.Point` 对象，它指定这个新窗口的左上角在屏幕中的位置。在图形用户界面程序中，通常(0,0)是显示器的左上角。这个默认的(-1,-1)将让系统决定窗口的位置。

`size` ：一个 `wx.Size` 对象，它指定这个窗口的初始尺寸。这个默认的(-1,-1)将让系统决定窗口的初始尺寸。

`style` ：指定窗口的类型的常量。你可以使用或运算来组合它们。

`name` ：框架的内在的名字。以后你可以使用它来寻找这个窗口。

记住，这些参数将被传递给父类的构造器方法： `wx.Frame.__init__()` 。

创建 `wx.Frame` 子类的方法如下所示：

```
class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "My Friendly Window",
            (100, 100), (100, 100))
```

使用wxPython的ID

在 `wxPython` 中， `ID` 号是所有窗口部件的特征。在一个 `wxPython` 应用程序中，每个窗口部件都有一个窗口标识。在每一个框架内， `ID` 号必须是唯一的，但是在框架之间你可以重用 `ID` 号。然而，我们建议你在你的整个应用程序中保

持 `ID` 号的唯一性，以防止处理事件时产生错误和混淆。在 `wxPython` 中也有一些标准的预定义的 `ID` 号，它们有特定的意思（例如， `wx.ID_OK` 和 `wx.ID_CANCEL` 是对话框中的 `OK` 和 `Cancel` 按钮的 `ID` 号）。在你的应用程序中重用标准的 `ID` 号一般没什么问题，只要你在预期的方式中使用它们。在 `wxPython` 中， `ID` 号的最重要的用处是在指定的对象发生的事件和响应该事件的回调函数之间建立唯一的关联。

有三种方法来创建一个窗口部件使用的 `ID` 号：

1、明确地给构造器传递一个正的整数 2、使用 `wx.NewId()` 函数 3、传递一个全局常量 `wx.ID_ANY` 或-1给窗口部件的构造器

明确地选择 **`ID`** 号

第一个或最直接的方法是明确地给构造器传递一个正的整数作为该窗口部件的 `ID` 。如果你这样做了，你必须确保在一个框架内没有重复的 `ID` 或重用了一个预定义的常量。你可以通过调用 `wx.RegisterId()` 来确保在应用程序中 `wxPython` 不在别处使用你的 `ID` 。要防止你的程序使用相同的 `wxPython ID` ，你应该避免使用全局常量 `wx.ID_LOWEST` 和 `wx.ID_HIGHEST` 之间的 `ID` 号。

使用全局性的 **`NewID()`** 函数

自己确保 `ID` 号的唯一性十分麻烦，你可以使用 `wx.NewId()` 函数让 `wxPython` 来为你创建 `ID` ：

```
id = wx.NewId()
frame = wx.Frame.__init__(None, id)
```

你也可以给窗口部件的构造器传递全局常量 **`wx.ID_ANY`** 或**-1**，然后 `wxPython` 将为你生成新的 `ID` 。然后你可以在需要这个 `ID` 时使用 `GetId()` 方法来得到它：

```
frame = wx.Frame.__init__(None, -1)
id = frame.GetId()
```

使用wx.Size和wx.Point

`wx.Frame` 构造器的参数也引用了类 `wx.Size` 和 `wx.Point` 。这两个类在你的 `wxPython` 编程中将频繁被使用。 `wx.Point` 类表示一个点或位置。构造器要求点的x和y值。如果不设置x,y值，则值默认为0。我们可以使用 `Set(x，y)` 和 `Get()` 函数来设置和得到x和y值。 `Get()` 函数返回一个元组。x和y值可以像下面这样作为属性被访问：

```
point = wx.Point(10, 12)
x = point.x
y = point.y
```

另外，`wx.Point` 的实例可以像其它 `Python` 对象一样作加、减和比较运算，例如：

```
a = wx.Point(2, 3)
b = wx.Point(5, 7)
c = a + b
bigger = a > b
```

在 `wx.Point` 的实参中，坐标值一般为整数。如果你需要浮点数坐标，你可以使用类 `wx.RealPoint` ，它的用法如同 `wx.Point` 。

`wx.Size` 类几乎和 `wx.Point` 完全相同，除了实参的名字是 `width` 和 `height` 。对 `wx.Size` 的操作与 `wx.Point` 一样。

在你的应用程序中当一个 `wx.Point` 或 `wx.Size` 实例被要求的时候（例如在另一个对象的构造器中），你不必显式地创建这个实例。你可以传递一个元组给构造器， `wxPython` 将隐含地创建这个 `wx.Point` 或 `wx.Size` 实例：

```
frame = wx.Frame(None, -1, pos=(10, 10), size=(100, 100))
```

使用wx.Frame的样式

每个 `wxPython` 窗口部件都要求一个样式参数。这部分我们将讨论用于 `wx.Frame` 的样式。它们中的一些也适用于别的 `wxPython` 窗口部件。一些窗口部件也定义了一个 `SetStyle()` 方法，让你可以在该窗口部件创建后改变它的样式。所有的你能使用的样式元素都有一个常量标识符（如 `wx.MINIMIZE_BOX` ）。要使用多个样式，你可以使用或运算符|。例如， `wx.DEFAULT_FRAME_STYLE` 样式就被定义为如下几个基本样式的组合：

```
wx.MAXIMIZE_BOX | wx.MINIMIZE_BOX | wx.RESIZE_BORDER |wx.SYSTEM_
MENU | wx.CAPTION | wx.CLOSE_BOX
```

要从一个合成的样式中去掉个别的样式，你可以使用^操作符。例如要创建一个默认样式的窗口，但要求用户不能缩放和改变窗口的尺寸，你可以这样做：

```
wx.DEFAULT_FRAME_STYLE ^ (wx.RESIZE_BORDER | wx.MINIMIZE_BOX |wx
.MAXIMIZE_BOX)
```

如果你不慎使用了&操作符，那么将得到一个没有样式的、无边框图的、不能移动、不能改变尺寸和不能关闭的帧。

下表**2.2**列出了用于 `wx.Frame` 的最重要的样式：

`wx.CAPTION` ：在框架上增加一个标题栏，它显示该框架的标题属性。 `wx.CLOSE_BOX` ：指示系统在框架的标题栏上显示一个关闭框，使用系统默认的位置和样式。 `wx.DEFAULT_FRAME_STYLE` ：默认样式。 `wx.FRAME_SHAPED` ：用这个样式创建的框架可以使用 `SetShape()` 方法去创建一个非矩形的窗口。 `wx.FRAME_TOOL_WINDOW` ：通过给框架一个比正常更小的标题栏，使框架看起来像一个工具框窗口。在 `Windows` 下，使用这个样式创建的框架不会出现在显示所有打开窗口的任务栏上。 `wx.MAXIMIZE_BOX` ：指示系统在框架的标题栏上显示一个最大化框，使用系统默认的位置和样式。 `wx.MINIMIZE_BOX` ：指示系统在框架的标题栏上显示一个最小化框，使用系统默认的位置和样式。 `wx.RESIZE_BORDER` ：给框架增加一个可以改变尺寸的边框。 `wx.SIMPLE_BORDER` ：没有装饰的边框。不能工作在所有平台上。 `wx.SYSTEM_MENU` ：增加系统菜单（带有关闭、移动、改变尺寸等功能）和关闭框到这个窗口。在系统菜单中的改变尺寸和关闭功能的有效性依赖于 `wx.MAXIMIZE_BOX` ， `wx.MINIMIZE_BOX` 和 `wx.CLOSE_BOX` 样式是否被应用。

下面的四张图显示了几个通常的框架的样式。

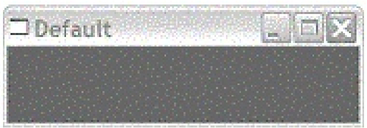


Figure 2.4 A frame created with the default style

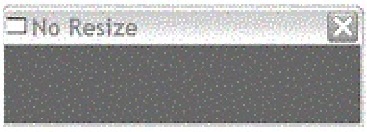


Figure 2.5 A frame created to be non-resizable. Notice the lack of minimize/maximize buttons.



Figure 2.6 A toolbar frame, with a smaller title bar and no system menu

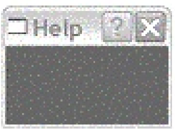


Figure 2.7 A frame with a help button

图2.4是使用 `wx.DEFAULT_STYLE` 创建的。 图2.5是使用 `wx.DEFAULT_FRAME_STYLE` ^(`wx.RESIZE_BORDER` | `wx.MINIMIZE_BOX` | `wx.MAXIMIZE_BOX)` 组合样式创建的。 图2.6使用的样式是 `wx.DEFAULT_FRAME_STYLE` | `wx.FRAME_TOOL_WINDOW` 。 图2.7使用了扩展样式 `wx.help.FRAME_EX_CONTEXTHELP` 。

如何为一个框架增加对象和子窗口?

我们已经说明了如何创建 `wx.Frame` 对象，但是创建后的是空的。本节我们将介绍在你的框架中插入对象与子窗口的基础，以便与用户交互。

给框架增加窗口部件

图2.8显示了一个定制的 `wx.Frame` 的子类，名为 `InsertFrame` 。当点击 `close` 按钮时，这个窗口将关闭且应用程序将退出。例2.3定义了子类 `InsertFrame` 。

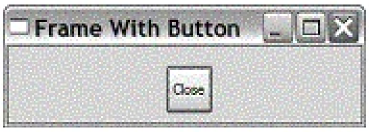


Figure 2.8 The `InsertFrame` window is an example demonstrating the basics of inserting items into a frame.

例2.3

```
#!/usr/bin/env python

import wx

class InsertFrame(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Frame With Button',
                size=(300, 100))
        panel = wx.Panel(self) #创建画板
        button = wx.Button(panel, label="Close", pos=(125, 10),
                size=(50, 50)) #将按钮添加到画板
#绑定按钮的单击事件
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
#绑定窗口的关闭事件
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

    def OnCloseMe(self, event):
        self.Close(True)

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = InsertFrame(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

类 `InsertFrame` 的方法 `__init__` 创建了两子窗口。第一个是 `wx.Panel` ，它是其它窗口的容器，它自身也有一点功能。第二个是 `wx.Button` ，它是一个平常按钮。接下来，按钮的单击事件和窗口的关闭事件被绑定到了相应的函数，当事件发生时这相应的函数将被调用执行。

大多数情况下，你将创建一个与你的 `wx.Frame` 大小一样的 `wx.Panel` 实例以容纳你的框架上的所有的内容。这样做可以让定制的窗口内容与其他如工具栏和状态栏分开。 通过 `tab` 按钮，可以遍历 `wx.Panel` 中的元素， `wx.Frame` 不能。

你不必像使用别的 `UI` 工具包那样，你不需要显式调用一个增加方法来向双亲中插入一个子窗口。在 `wxPython` 中，你只需在子窗口被创建时指定父窗口，这个子窗口就隐式地增加到父对象中了，例如例2.3所示。

你可能想知道在例2.3中，为什么 `wx.Button` 被创建时使用了明确的位置和尺寸，而 `wx.Panel` 没有。在 `wxPython` 中，如果只有一个子窗口的框架被创建，那么那个子窗口（例2.3中是 `wx.Panel` ）被自动重新调整尺寸去填满该框架的客户区域。这个自动调整尺寸将覆盖关于这个子窗口的任何位置和尺寸信息，尽管关于子窗口的信息已被指定，这些信息将被忽略。这个自动调整尺寸仅适用于框架内或对

话框内的只有唯一元素的情况。这里按钮是 `panel` 的元素，而不是框架的，所以要使用指定的尺寸和位置。如果没有为这个按钮指定尺寸和位置，它将使用默认的位置（ `panel` 的左上角）和基于按钮标签的长度的尺寸。

显式地指定所有子窗口的位置和尺寸是十分乏味的。更重要的是，当用户调整窗口大小的时候，这使得子窗口的位置和大小不能作相应调整。为了解决这两个问题， `wxPython` 使用了称为 `sizers` 的对象来管理子窗口的复杂布局。

给框架增加菜单栏、工具栏和状态栏。

图2.9显示了一个有菜单栏、工具栏和状态栏的框架。

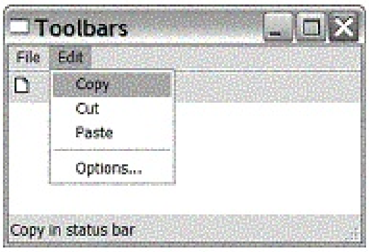


Figure 2.9 A sample frame with menubar, toolbar, and status bar

例2.4显示了 `__init__` 方法，它用这三个子窗口装饰了一个简单的窗口。 例2.4

```
#!/usr/bin/env python

import wx
import images

class ToolbarFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Toolbars',
                size=(300, 200))
        panel = wx.Panel(self)
        panel.SetBackgroundColour('White')
        statusBar = self.CreateStatusBar() #1 创建状态栏
        toolbar = self.CreateToolBar()     #2 创建工具栏
        toolbar.AddSimpleTool(wx.NewId(), images.getNewBitmap(),
                "New", "Long help for 'New'") #3 给工具栏增加一个工
具
        toolbar.Realize() #4 准备显示工具栏
        menuBar = wx.MenuBar() # 创建菜单栏
# 创建两个菜单
        menu1 = wx.Menu()
        menuBar.Append(menu1, "   ")
        menu2 = wx.Menu()
#6 创建菜单的项目
        menu2.Append(wx.NewId(), "   ", "Copy in status bar")
        menu2.Append(wx.NewId(), "C ", "")
        menu2.Append(wx.NewId(), "Paste", "")
        menu2.AppendSeparator()
        menu2.Append(wx.NewId(), "   ", "Display Options")
        menuBar.Append(menu2, "   ") # 在菜单栏上附上菜单
        self.SetMenuBar(menuBar)  # 在框架上附上菜单栏

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ToolbarFrame(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

说明：

- **#1**：这行创建了一个状态栏，它是类 `wx.StatusBar` 的实例。它被放置在框架的底部，宽度与框架相同，高度由操作系统决定。状态栏的目的是显示在应用程序中被各种事件所设置的文本。

  #2：创建了一个 `wx.ToolBar` 的实例，它是命令按钮的容器。它被自动放置在框架的顶部

  #3：有两种方法来为你工具栏增加工具，这行使用了参数较少的一种： `AddSimpleTool()` 。参数分别是 `ID` ，位图，该工具的短的帮助提示文本，显示在状态栏中的该工具的长的帮助文本信息。（此刻不要考虑位图从哪儿来）

#4：`Realize()` 方法告诉工具栏这些工具按钮应该被放置在哪儿。这是必须的。

#6：创建菜单的项目，其中参数分别代表 `ID` ，选项的文本，当鼠标位于其上时显示在状态栏的文本。

如何使用一般的对话框?

`wxPython` 提供了一套丰富的预定义的对话框。这部分，我们将讨论三种用对话框得到来自用户的信息：

1、消息对话框 2、文本输入对话框 3、从一个列表中选择

在 `wxPython` 中有许多别的标准对话框，包括文件选择器、色彩选择器、进度对话框、打印设置和字体选择器。这些将在第9章介绍。

消息对话框

与用户通信最基本的机制是 `wx.MessageDialog` ，它是一个简单的提示框。 `wx.MessageDialog` 可用作一个简单的 `OK` 框或 `yes` / `no` 对话框。下面的片断显示了 `yes` / `no` 对话框：

```
dlg = wx.MessageDialog(None, 'Is this the coolest thing ever!',
                       'MessageDialog', wx.YES_NO | wx.ICON_QUEST
ION)
result = dlg.ShowModal()
dlg.Destroy()
```

显示结果如图2.10所示：



Figure 2.10 A message dialog, configured for a yes/no response

`wx.MessageDialog` 参数如下：

```
wx.MessageDialog(parent, message,
        caption="Message box",
        style=wx.OK | wx.CANCEL,
        pos=wx.DefaultPosition)
```

参数说明：

`parent` ：对话框的父窗口，如果对话框是顶级的则为 `None` 。 `message` ：显示在对话框中的字符串。 `caption` ：显示在对话框标题栏上的字符串。 `style` ：对话框中按钮的样式。 `pos` ：对话框出现的位置。

`ShowModal()` 方法将对话框以模式框架的方式显示，这意味着在对话框关闭之前，应用程序中的别的窗口不能响应用户事件。 `ShowModal()` 方法的返回值是一个整数，对于 `wx.MessageDialog` ，返回值是下面常量之一： `wx.ID_YES` , `wx.ID_NO` , `wx.ID_CANCEL` , `wx.ID_OK` 。

文本输入对话框

如果你想从用户那里得到单独一行文本，你可能使用类 `wx.TextEntryDialog` 。下面的片断创建了一个文本输入域，当用户单击 `OK` 按钮退出时，获得用户输入的值：

```
dlg = wx.TextEntryDialog(None, "Who is buried in Grant's tomb?",
        'A Question', 'Cary Grant')
if dlg.ShowModal() == wx.ID_OK:
    response = dlg.GetValue()
```

图2.11显示了上面这个对话框：

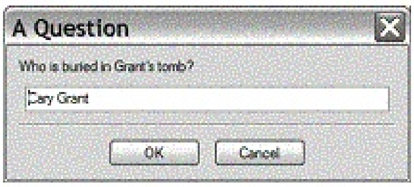


Figure 2.11 A text entry dialog

上面的 `wx.TextEntryDialog` 的参数按顺序说明是，父窗口，显示在窗口中的文本标签，窗口的标题（默认是“ `Please enter text` ”），输入域中的默认值。同样它也有一个样式参数，默认是 `wx.OK | wx.CANCEL` 。与 `wx.MessageDialog` 一样， `ShowModal()` 方法返回所按下的按钮的 `ID` 。 `GetValue()` 方法得到用户输入在文本域中的值（这有一个相应的 `SetValue()` 方法让你可以改变文本域中的值）。

从一个列表中选择

你可以让用户只能从你所提供的列表中选择，你可以使用类 `wx.SingleChoiceDialog` 。下面是一个简单的用法：

```
dlg = wx.SingleChoiceDialog(None,
        'What version of Python are you using?',
        'Single Choice',
        ['1.5.2', '2.0', '2.1.3', '2.2', '2.3.1'],
if dlg.ShowModal() == wx.ID_OK:
    response = dlg.GetStringSelection()
```

图2.12显示了上面代码片断的结果。

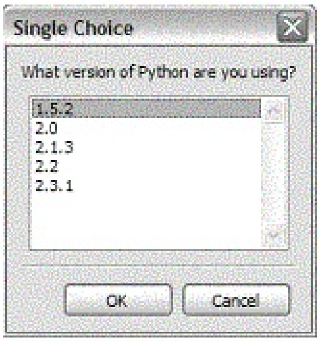


Figure 2.12 The SingleChoiceDialog window, allowing a user to choose from a predefined list

wx.SingleChoiceDialog 的参数类似于文本输入对话框，只是以字符串的列表代替了默认的字符串文本。要得到所选择的结果有两种方法， GetSelection() 方法返回用户选项的索引，而 GetStringSelection() 返回实际所选的字符串。

一些最常见的错误现象及解决方法?

有一些错误它们可能会发生在你的 wxPython 应用程序对象或初始的顶级窗口在创建时，这些错误可能是很难诊断的。下面我们列出一些最常见的错误现象及解决方法：

错误现象：程序启动时提示“ unable to import module wx 。”

原因： wxPython 模块不在你的 PYTHONPATH 中。这意味着 wxPython 没有被正确地安装。如果你的系统上有多个版本的 Python ， wxPython 可能安装在了你没有使用的 Python 版本中。

解决方法：首先，确定你的系统上安装了哪些版本的 Python 。在 Unix 系统上，使用 which python 命令将告诉你默认的安装。在 Windows 系统上，如果 wxPython 被安装到了相应的 Python 版本中，它将位于 python - home / Lib / site - packages 子目录下。然后重装 wxPython 。

错误现象：应用程序启动时立即崩溃，或崩溃后出现一空白窗口。

原因：在 wx.App 创建之前，创建或使用了一个 wxPython 对象。

解决方法：在启动你的脚本时立即创建 wx.App 对象。

错误现象：顶级窗口被创建同时又立刻关闭。应用程序立即退出。

原因：没有调用 wx.App 的 MainLoop() 方法。

解决方法：在你的所有设置完成后调用 MainLoop() 方法。

错误现象：顶级窗口被创建同时又立刻关闭。应用程序立即退出。但我调用了 `MainLoop()` 方法。

原因：你的应用程序的 `OnInit()` 方法中有错误，或 `OnInit()` 方法调用了某些方法（如帧的 `__init__()` 方法）。

解决方法：在 `MainLoop()` 被调用之前出现错误的话，这将触发一个异常且程序退出。如果你的应用程序设置了重定向输出到窗口，那么那些窗口将一闪而过，你不能看到显示在窗口中的错误信息。这种情况下，你要使用 `redirect` = `False` 关闭重定向选项，以便看到错误提示。

总结

1. `wxPython` 程序的实现基于两个必要的对象：应用程序对象和顶级窗口。任何 `wxPython` 应用程序都需要去实例化一个 `wx.App` ，并且至少有一个顶级窗口。

2. 应用程序对象包含 `OnInit()` 方法，它在启动时被调用。在这个方法中，通常要初始化框架和别的全局对象。 `wxPython` 应用程序通常在它的所有的顶级窗口被关闭或主事件循环退出时结束。

3. 应用程序对象也控制 `wxPython` 文本输出的位置。默认情况下， `wxPython` 重定向 `stdout` 和 `stderr` 到一个特定的窗口。这个行为使得诊断启动时产生的错误变得困难了。但是我们可以通过让 `wxPython` 把错误消息发送到一个文件或控制台窗口来解决。

4. 一个 `wxPython` 应用程序通常至少有一个 `wx.Frame` 的子类。一个 `wx.Frame` 对象可以使用 `style` 参数来创建组合的样式。每个 `wxWidget` 对象，包括框架，都有一个 `ID` ，这个 `ID` 可以被应用程序显式地赋值或由 `wxPython` 生成。子窗口是框架的内容，框架是它的双亲。通常，一个框架包含一个单一的 `wx.Panel` ，更多的子窗口被放置在这个 `Panel` 中。框架的唯一的子窗口的尺寸自动随其父框架的尺寸的改变而改变。框架有明确的关于管理菜单栏、工具栏和状态栏的机制。

5. 尽管你将使用框架做任何复杂的事情，但当你想简单而快速地得到来自用户的信息时，你可以给用户显示一个标准的对话窗口。对于很多任务都有标准的对话框，包括警告框、简单的文本输入框和列表选择框等等。

第三章 在事件驱动环境中开发

事件处理是 `wxPython` 程序工作的基本机制。主要执行事件处理的工作称为事件驱动。在这章中我们将讨论什么是事件驱动应用程序，它与传统的应用程序有什么不同。我们将对在 `GUI` 编程中所使用的概念和术语提供一些介绍，包括与用户交互，工具包和编程逻辑。也将包括典型事件驱动程序的生命周期。

事件就是发生在你的系统中的事，你的应用程序通过触发相应的功能以响应它。事件可以是低级的用户动作，如鼠标移动或按键按下，也可以是高级的用户动作（定义在 `wxPython` 的窗口部件中的），如单击按钮或菜单选择。事件可以产生自系统，如关机。你甚至可以创建你自己的对象去产生你自己的事件。 `wxPython` 应用程序通过将特定类型的事件和特定的一块代码相关联来工作，该代码在响应事件时执行。事件被映射到代码的过程称为事件处理。

本章将说明事件是什么，你如何写响应一个事件的代码，以及 `wxPython` 在事件发生的时候是如何知道去调用你的代码的。我们也将说明如何将定制的事件增加到 `wxPython` 库中，该库包含了关于用户和系统行为的标准事件的一个列表。

要理解事件，我们需要知道哪些术语？

本章包含了大量的术语，很多都是以 `event` 开头的。下表3.1是我们将要用到的术语的一个快速参考：

事件(`event)` ：在你的应用程序期间发生的事情，它要求有一个响应。

事件对象(`event` `object)` ：在 `wxPython` 中，它具体代表一个事件，其中包括了事件的数据等属性。它是类 `wx.Event` 或其子类的实例，子类如 `wx.CommandEvent` 和 `wx.MouseEvent` 。

事件类型(`event` `type`) ： `wxPython` 分配给每个事件对象的一个整数 `ID` 。事件类型给出了关于该事件本身更多的信息。例如， `wx.MouseEvent` 的事件类型标识了该事件是一个鼠标单击还是一个鼠标移动。

事件源(`event` `source`) ：任何 `wxPython` 对象都能产生事件。例如按钮、菜单、列表框和任何别的窗口部件。

事件驱动(`event` - `driven`) ：一个程序结构，它的大部分时间花在等待或响应事件上。

事件队列(`event` `queue`) ：已发生的但未处理的事件的一个列表。

事件处理器(`event` `handler`) ：响应事件时所调用的函数或方法。也称作处理器函数或处理器方法。

事件绑定器(`event` `binder`) ：一个封装了特定窗口部件，特定事件类型和一个事件处理器的 `wxPython` 对象。为了被调用，所有事件处理器必须用一个事件绑定器注册。

`wx.EvtHandler` ：一个 `wxPython` 类，它允许它的实例在一个特定类型，一个事件源，和一个事件处理器之间创建绑定。注意，这个类与先前定义的事件处理函数或方法不是同一个东西。

什么是事件驱动编程?

事件驱动程序主要是一个控制结构，它接受事件并响应它们。 `wxPython` 程序（或任何事件驱动程序）的结构与平常的 `Python` 脚本不同。标准的 `Python` 脚本有一个特定的开始点和结束点，程序员使用条件、循环、和函数来控制执行顺序。

从用户的角度上来看， `wxPython` 程序大部分时间什么也不做，一直闲着直到用户或系统做了些什么来触发这个 `wxPython` 程序动作。 `wxPython` 程序的结构就是一个事件驱动程序体系的例子。图3.1是事件处理循环的示意，它展示了主程序的生命、用户事件、和分派到的处理器函数。

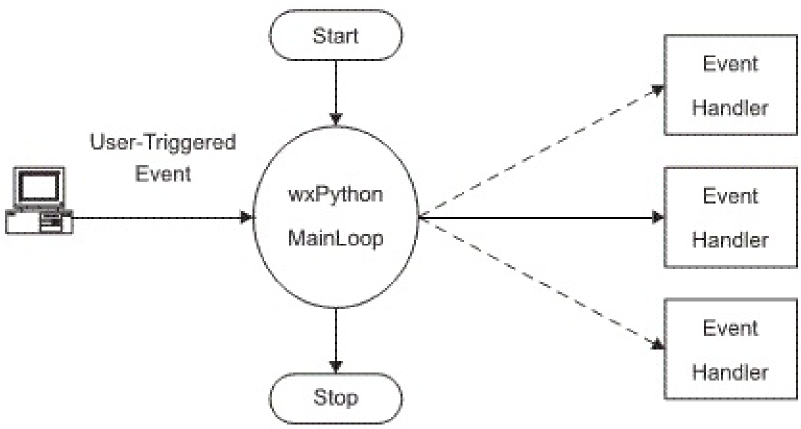


Figure 3.1 A schematic of the event handling cycle, showing the life of the main program, a user event, and dispatch to handler functions.

事件驱动系统的主循环类似于客户服务呼叫中心的操作者。当没有呼叫的进入的时候，这个操作者处于等待状态。当一个事件发生的时候，如电话铃响了，这个操作者开始一个响应过程，他与客户交谈直到他获得足够的信息以分派该客户给一个合适的回答者。然后操作者等待下一个事件。

尽管每个事件驱动系统之间有一些不同，但它们有很多相似的地方。下面列出了事件驱动程序结构的主要特点：

1、在初始化设置之后，程序的大部分时间花在了一个空闭的循环之中。进入这个循环就标志着程序与用户交互的部分的开始，退出这个循环就标志结束。在 `wxPython` 中，这个循环的方法是： `wx.App.MainLoop()` ，并且在你的脚本中显式地被调用。当所有的顶级窗口关闭时，主循环退出。

2、程序包含了对应于发生在程序环境中的事情的事件。事件通常由用户的行为触发，但是也可以由系统的行为或程序中其他任意的代码。在 `wxPython` 中，所有的事件都是类 `wx.Event` 或其子类的一个实例。每个事件都有一个事件类型属性，它使得不同的事件能够被辨别。例如，鼠标释放和鼠示按下事件都被认为是同一个类的实例，但有不同的事件类型。

3、作为这个空闭的循环部分，程序定期检查是否有任何请求响应事情发生。有两种机制使得事件驱动系统可以得到有关事件的通知。最常被 `wxPython` 使用的方法是，把事件传送到一个中心队列，由该队列触发相应事件的处理。另一种方法是使用轮询的方法，所有可能引发事件的事件主被主过程定期查询并询问是否有没有处理的事件。

4、当事件发生时，基于事件的系统试着确定相关代码来处理该事件，如果有，相关代码被执行。在 `wxPython` 中，原系统事件被转换为 `wx.Event` 实例，然后使用 `wx.EvtHandler.ProcessEvent()` 方法将事件分派给适当的处理器代码。图3.3呈现了这个过程：

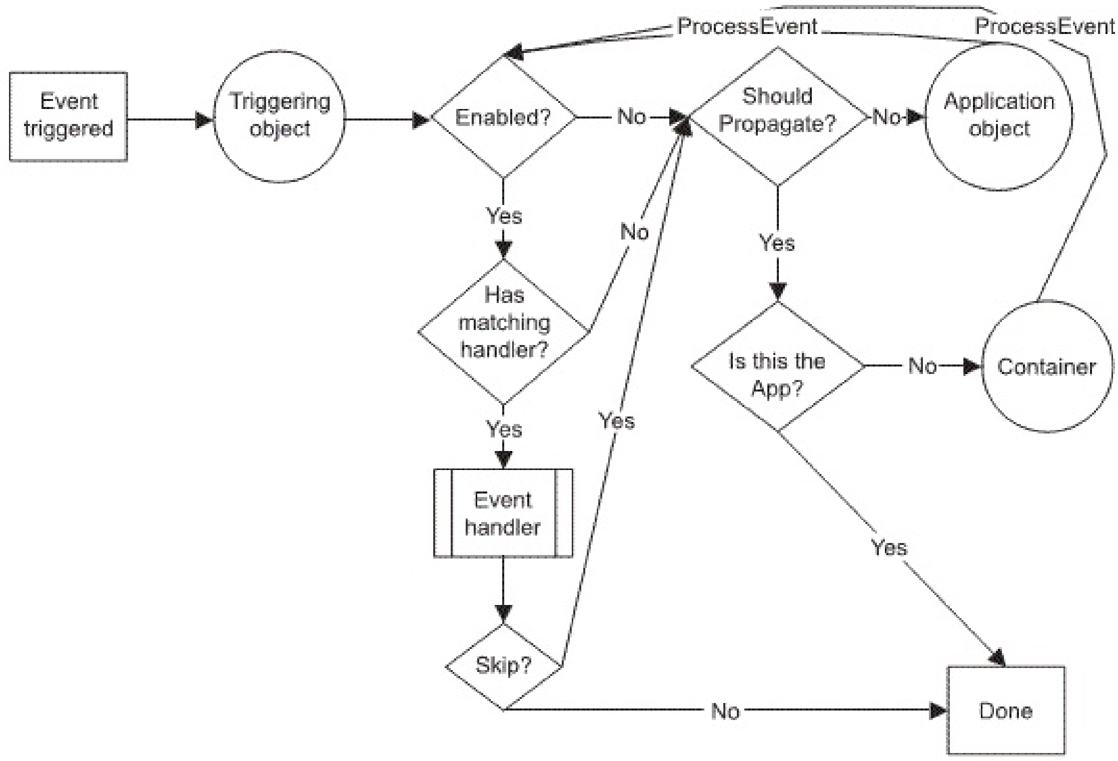


Figure 3.3 Event handling process, starting with the event being triggered, and moving through the steps of searching for a handler

事件机制的组成部分是事件绑定器对象和事件处理器。事件绑定器是一个预定义的 `wxPython` 对象。每个事件都有各自的事件绑定器。事件处理器是一个函数或方法，它要求一个 `wxPython` 事件实例作为参数。当用户触发了适当的事件时，一个事件处理器被调用。

下面我们将讨论有关 `wxPython` 更多的细节，我们把事件响应的基本单元“事件处理器”作为开始。

编写事件处理器

在你的 `wxPython` 代码中，事件和事件处理器是基于相关的窗口部件的。例如，一个按钮被单击被分派给一个基于该按钮的专用的事件处理器。为了要把一个来自特定窗口部件的事件绑定到一个特定的处理器方法，你要使用一个绑定器对象来管理这个连接。例如：

```
self.Bind(wx.EVT_BUTTON, self.OnClick, aButton)
```

上例使用了预定义的事件绑定器对象 `wx.EVT_BUTTON` 来将 `aButton` 对象上的按钮单击事件与方法 `self.OnClick` 相关联起来。这个 `Bind()` 方法是 `wx.EvtHandler` 的一个方法， `wx.EvtHandler` 是所有可显示对象的父类。因此上例代码行可以被放置在任何显示类。

即使你的 `wxPython` 程序表面上看起来在被动地等待事件，但它仍在做事。它在运行方法 `wx.App.MainLoop()` ，该方法是一个无限的循环。 `MainLoop()` 方法可以使用 `Python` 伪代码表示如下：

```
while True:
    while not self.Pending():
self.ProcessIdle()
self.DoMessage()
```

上面的伪代码意思是如果没有未处理的消息，则做一些空闲时做的事；如果有消息进入，那么将这个消息分派给适当的事件处理方法。

设计事件驱动程序

对于事件驱动程序的设计，由于没有假设事件何时发生，所以程序员将大量的控制交给了用户。你的 `wxPython` 程序中的大多数代码通过用户或系统的行为被直接或间接地执行。例如在用户选择了一个菜单项、或按下一个工具栏按钮、或按下了特定的按键组合后，你的程序中有关保存工作的代码被执行了。

另一方面，事件驱动体系通常是分散性的。响应一个窗口部件事件的代码通常不是定义在该部件的定义中的。例如，响应一个按钮单击事件的代码不必是该按钮定义的一部分，而可以存在在该按钮所附的框架中或其它地方。当与面向对象设计结合时，这个体系导致了松散和高度可重用的代码。你将会发现 `Python` 的灵活使得重用不同的 `wxPython` 应用程序的通常的事件处理器和结构变得非常容易。

事件触发

在 `wxPython` 中，大部分窗口部件在响应低级事件时都导致高级事件发生。例如，在一个 `wx.Button` 上的鼠标单击导致一个 `EVT_BUTTON` 事件的生成，该事件是 `wx.CommandEvent` 的特定类型。类似的，在一个窗口的角中拖动鼠标将导致 `wxPython` 为你自动创建一个 `wx.SizeEvent` 事件。高级事件的用处是让你的系统的其它部分更容易聚焦于最有关联的事件上，而不是陷于追踪每个鼠标单击。高级事件能够封装更多关于事件的有用的信息。当你创建你自己的定制的窗口部件时，你能定义你自己的定制事件以便管理事件的处理。

在 `wxPython` 中，代表事件的是事件对象。事件对象是类 `wx.Event` 或其子类的一个实例。父类 `wx.Event` 相对小且抽象，它只是包含了对所有事件的一些通常的信息。 `wx.Event` 的各个子类都添加了更多的信息。

在 `wxPython` 中，有一些 `wx.Event` 的子类。表3.2包含了你将最常遇到的一些事件类。记住，一个事件类可以有多个事件类型，每个都对应于一个不同的用户行为。下表3.2是 `wx.Event` 的重要的子类。

`wx.CloseEvent` ：当一个框架关闭时触发。这个事件的类型分为一个通常的框架关闭和一个系统关闭事件。 `wx.CommandEvent` ：与窗口部件的简单的各种交互都将触发这个事件，如按钮单击、菜单项选择、单选按钮选择。这些交互有它各自的

事件类型。许多更复杂的窗口部件，如列表等则定义 `wx.CommandEvent` 的子类。事件处理系统对待命令事件与其它事件不同。 `wx.KeyEvent` ：按按键事件。这个事件的类型分按下按键、释放按键、整个按键动作。 `wx.MouseEvent` ：鼠标事件。这个事件的类型分鼠标移动和鼠标敲击。对于哪个鼠标按钮被敲击和是单击还是双击都有各自的事件类型。 `wx.PaintEvent` ：当窗口的内容需要被重画时触发。 `wx.SizeEvent` ：当窗口的大小或其布局改变时触发。 `wx.TimerEvent` ：可以由类 `wx.Timer` 类创建，它是定期的事件。

通常，事件对象需要使用事件绑定器和事件处理系统将它们传递给相关的事件处理器。

如何将事件绑定到处理器?

事件绑定器由类 `wx.PyEventBinder` 的实例组成。一个预定义的 `wx.PyEventBinder` 的实例被提供给所有支持的事件类型，并且在你需要的时候你可以为你定制的事件创建你自己的事件绑定器。每个事件类型都有一个事件绑定器，这意味着一个 `wx.Event` 的子类对应多个绑定器。

在 `wxPython` 中，事件绑定器实例的名字是全局性的。为了清楚地将事件类型与处理器联系起来，它们的名字都是以 `wx.EVT_` 开头并且对应于使用在C++ `wxWidgets` 代码中宏的名字。值得强调的是， `wx.EVT` 绑定器名字的值不是你通过调用一个 `wx.Event` 实例的 `GetEventType()` 方法得到的事件类型的实际的整数码。事件类型整数码有一套完全不同的全局名，并且在实际中不常被使用。

作为 `wx.EVT` 名字的例子，让我们看看 `wx.MouseEvent` 的事件类型。正如我们所提到的，它们有十四个，其中的九个涉及到了基于在按钮上的敲击，如鼠标按下、鼠标释放、或双击事件。这九个事件类型使用了下面的名字：

```
wx.EVT_LEFT_DOWN
wx.EVT_LEFT_UP
wx.EVT_LEFT_DCLICK
wx.EVT_MIDDLE_DOWN
wx.EVT_MIDDLE_UP
wx.EVT_MIDDLE_DCLICK
wx.EVT_RIGHT_DOWN
wx.EVT_RIGHT_UP
wx.EVT_RIGHT_DCLICK
```

另外，类型 `wx.EVT_MOTION` 产生于用户移动鼠标。类型 `wx.ENTER_WINDOW` 和 `wx.LEAVE_WINDOW` 产生于当鼠标进入或离开一个窗口部件时。类型 `wx.EVT_MOUSEWHEEL` 被绑定到鼠标滚轮的活动。最后，你可以使用类型 `wx.EVT_MOUSE_EVENTS` 一次绑定所有的鼠标事件到一个函数。

同样，类 `wx.CommandEvent` 有28个不同的事件类型与之关联；尽管有几个仅针对老的 `Windows` 操作系统。它们中的大多数是专门针对单一窗口部件的，如 `wx.EVT_BUTTON` 用于按钮敲击， `wx.EVT_MENU` 用于菜单项选择。用于专门窗口部件的命令事件在 `part2` 中讨论。

绑定机制的好处是它使得 `wxPython` 可以很细化地分派事件，而仍然允许同类的类似事件发生并且共享数据和功能。这使得在 `wxPython` 中写事件处理比在其它界面工具包中清细得多。

事件绑定器被用于将一个 `wxPython` 窗口部件与一个事件对象和一个处理器函数连接起来。这个连接使得 `wxPython` 系统能够通过执行处理器函数中的代码来响应相应窗口部件上的事件。在 `wxPython` 中，任何能够响应事件的对象都是 `wx.EvtHandler` 的子类。所有窗口对象都是 `wx.EvtHandler` 的子类，因此在 `wxPython` 应用程序中的每个窗口部件都能够响应事件。类 `wx.EvtHandler` 也能够被非窗口部件对象所使用，如 `wx.App` ，因此事件处理功能不是限于可显示的窗口部件。我们所说的窗口部件能响应事件的意思是：该窗口部件能够创建事件绑定，在分派期间 `wxPython` 能够识别该事件绑定。由绑定器调用的在事件处理器函数中的实际代码不是必须位于一个 `wx.EvtHandler` 类中。

使用**wx.EvtHandler**的方法工作

`wx.EvtHandler` 类定义的一些方法在一般情况下用不到。你会经常使用的 `wx.EvtHandler` 的方法是 `Bind()` ，它创建事件绑定。该方法的用法如下：

```
Bind(event, handler, source=None, id=wx.ID_ANY, id2=wx.ID_ANY)
```

`Bind()` 函数将一个事件和一个对象与一个事件处理器函数关联起来。参数 `event` 是必选的，它是我们在3.3节中所说的 `wx.PyEventBinder` 的一个实例。参数 `handler` 也是必选的，它是一个可调用的 `Python` 对象，通常是一个被绑定的方法或函数。处理器必须是可使用一个参数（事件对象本身）来调用的。参数 `handler` 可以是 `None` ，这种情况下，事件没有关联的处理器。参数 `source` 是产生该事件的源窗口部件，这个参数在触发事件的窗口部件与用作事件处理器的窗口部件不相同时使用。通常情况下这个参数使用默认值 `None` ，这是因为你一般使用一个定制的 `wx.Frame` 类作为处理器，并且绑定来自于包含在该框架内的窗口部件的事件。父窗口的 `__init__` 是一个用于声明事件绑定的方便的位置。但是如果父窗口包含了多个按钮敲击事件源（比如 `OK` 按钮和 `Cancel` 按钮），那么就要指定 `source` 参数以便 `wxPython` 区分它们。下面是该方法的一个例子：

```
self.Bind(wx.EVT_BUTTON, self.OnClick, button)
```

下例3.1演示了使用参数 `source` 和不使用参数 `source` 的方法，它改编自第二章中的代码：

```
def __init__(self, parent, id):
    wx.Frame.__init__(self, parent, id, 'Frame With Button',
size=(300, 100))
panel = wx.Panel(self, -1)
button = wx.Button(panel, -1, "Close", pos=(130, 15),
size=(40, 40))
    self.Bind(wx.EVT_CLOSE, self.OnCloseWindow) #1 绑定框架关闭事件
    self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button) #2 绑定按钮事
件

def OnCloseMe(self, event):
self.Close(True)
    def OnCloseWindow(self, event):
self.Destroy()
```

说明：

#1 这行绑定框架关闭事件到 `self.OnCloseWindow` 方法。由于这个事件通过该框架触发且用于帧，所以不需要传递一个 `source` 参数。

#2 这行将来自按钮对象的按钮敲击事件绑定到 `self.OnCloseMe` 方法。这样做是为了让 `wxPython` 能够区分在这个框架中该按钮和其它按钮所产生的事件。

你也可以使用 `source` 参数来标识项目，即使该项目不是事件的源。例如，你可以绑定一个菜单事件到事件处理器，即使这个菜单事件严格地说是由框架所触发的。下例3.2演示了绑定一个菜单事件的例子：

```
#!/usr/bin/env python

import wx

class MenuEventFrame(wx.Frame):
def __init__(self, parent, id):
wx.Frame.__init__(self, parent, id, 'Menus',
size=(300, 200))
menuBar = wx.MenuBar()
menu1 = wx.Menu()
menuItem = menu1.Append(-1, "  ")
menuBar.Append(menu1, "  ")
self.SetMenuBar(menuBar)
self.Bind(wx.EVT_MENU, self.OnCloseMe, menuItem)

def OnCloseMe(self, event):
self.Close(True)

if __name__ == '__main__':
app = wx.PySimpleApp()
frame = MenuEventFrame(parent=None, id=-1)
frame.Show()
app.MainLoop()
```

`Bind()` 方法中的参数 `id` 和 `id2` 使用 `ID` 号指定了事件的源。一般情况下这没必要，因为事件源的 `ID` 号可以从参数 `source` 中提取。但是某些时候直接使用 `ID` 是合理的。例如，如果你在使用一个对话框的 `ID` 号，这比使用窗口部件更容易。如果你同时使用了参数 `id` 和 `id2` ，你就能够以窗口部件的 `ID` 号形式将这两个 `ID` 号之间范围的窗口部件绑定到事件。这仅适用于窗口部件的 `ID` 号是连续的。

注意： `Bind()` 方法出现在 `wx.Python2.5` 中，以前版本的事件绑定中， `EVT_` *的用法如同函数对象，因此你会看到如下的绑定调用：

```
wx.EVT_BUTTON(self, self.button.GetId(), self.OnClick)
```

这个方式的缺点是它不像是面向对象的方法调用。然而，这个老的样式仍可工作在2.5的版本中（因为 `wx.EVT` *对象仍是可调用的）。

下表**3.3**列出了最常使用的 **`wx.EvtHandler`** 的方法：

`AddPendingEvent(event)` ：将这个 `event` 参数放入事件处理系统中。类似于 `ProcessEvent()` ，但它实际上不会立即触发事件的处理。相反，该事件被增加到事件队列中。适用于线程间的基于事件的通信。

```
Bind(event, handler, source=None,   id=wx.ID_ANY, id2=wx.ID_ANY)
 :
```

完整的说明见3.3.1节。

`GetEvtHandlerEnabled()` `SetEvtHandlerEnabled(` `boolean)` ：如果处理器当前正在处理事件，则属性为 `True` ，否则为 `False` 。

`ProcessEvent(event)` ：把 `event` 对象放入事件处理系统中以便立即处理。

wxPython是如何处理事件的?

基于事件系统的关键组成部分是事件处理。通过它，一个事件被分派到了相应的用于相应该事件的一块代码。在这一节，我们将讨论 `wxPython` 处理事件的过程。我们将使用小段的代码来跟踪这个处理的步骤。图3.2显示了一个带有一个按钮的简单窗口，这个按钮将被用来产生一个简单的事件。



Figure 3.2 A simple window with mouse events

下例3.3包含了生成这个窗口的代码。在这个代码中，通过敲击按钮和将鼠标移动到按钮上都可产生 `wxPython` 事件。

例3.3绑定多个鼠标事件

```
#!/usr/bin/env python

import wx

class MouseEventFrame(wx.Frame):

def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Frame With Button',
size=(300, 100))
self.panel = wx.Panel(self)
        self.button = wx.Button(self.panel,
label="Not Over", pos=(100, 15))
self.Bind(wx.EVT_BUTTON, self.OnButtonClick,
            self.button)    #1 绑定按钮事件
self.button.Bind(wx.EVT_ENTER_WINDOW,
            self.OnEnterWindow)     #2 绑定鼠标位于其上事件
self.button.Bind(wx.EVT_LEAVE_WINDOW,
            self.OnLeaveWindow)     #3 绑定鼠标离开事件

def OnButtonClick(self, event):
self.panel.SetBackgroundColour('Green')
self.panel.Refresh()

def OnEnterWindow(self, event):
self.button.SetLabel("Over Me!")
event.Skip()

def OnLeaveWindow(self, event):
self.button.SetLabel("Not Over")
event.Skip()

if __name__ == '__main__':
app = wx.PySimpleApp()
    frame = MouseEventFrame(parent=None, id=-1)
frame.Show()
app.MainLoop()
```

说明：

`MouseEventFrame` 包含了一个位于中间的按钮。在其上敲击鼠标将导致框架的背景色改变为绿色。#1绑定了鼠标敲击事件。当鼠标指针位于这个按钮上时，按钮上的标签将改变，这用#2绑定。当鼠标离开这个按钮时，标签变回原样，这用#3绑定。

通过观察上面的鼠标事件例子，我们引出了在 `wxPython` 中的事件处理的一些问题。#1中，按钮事件由附着在框架上的按钮触发，那么 `wxPython` 怎么知道在框架对象中查找绑定而不是在按钮对象上呢？在#2和#3中，鼠标的进入和离开事件被绑定到了按钮，为什么这两个事件不能被绑到框架上呢。这些问题将通过检查 `wxPython` 用来决定如何响应事件的过程来得到回答。

理解事件处理过程

`wxPython` 的事件处理过程被设计来简化程序员关于事件绑定的创建，使他们不必考虑哪些不重要的事件。 隐藏在简化设计之下的底层机制是有些复杂的。接下来，我们将跟踪关于按钮敲击和鼠标进入事件的过程。

图3.3显示了事件处理过程的一个基本的流程。矩形代表过程的开始和结束，环形代表各种 `wxPython` 对象（它们是这个过程的一部分），棱形代表判断点，带条的矩形代表实际的事件处理方法。

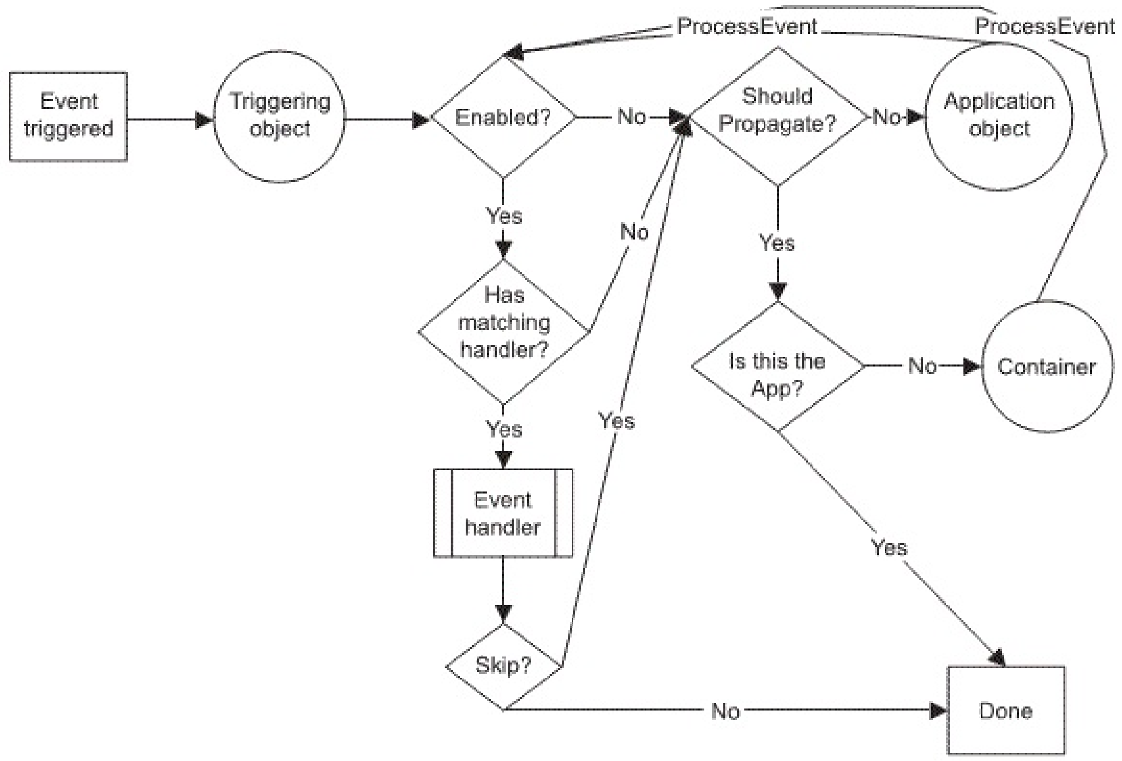


Figure 3.3 Event handling process, starting with the event being triggered, and moving through the steps of searching for a handler

事件处理过程开始于触发事件的对象。通常， `wxPython` 首先在触发对象中查找匹配事件类型的被绑定的处理器函数。如果找到，则相应的方法被执行。否则， `wxPython` 将检查该事件是否传送到了上一级的容器。如果是的话，父窗口部件将被检查，这样一级一级向上寻找，直到 `wxPython` 找到了一个处理器函数或到达了顶级窗口。如果事件没有传播，在处理过程完成之前， `wxPython` 仍将为了处理器函数而检查应用程序对象。

当事件处理器运行时，过程通常就结束了。然而，函数可以告诉 `wxPython` 去继续查找处理器。 下面让我们仔细观察一下这个过程的每一个步骤。我们的每步分析都有图3.3的一个相关略图。

第一步，创建事件

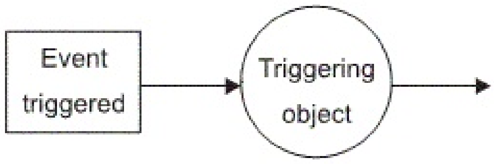


Figure 3.4 Creation of the event that sends focus to the triggering object

这个过程开始于事件被创建时。 在 `wxPython` 架构中已经创建了大多数的事件类型，它们用于响应特定的用户动作或系统通知。例如，当 `wxPython` 通知“鼠标移进了一个新窗口部件对象时”，鼠标进入事件被触发，鼠标敲击事件在鼠标按下或释放后被创建。

事件首先被交给创建事件的对象。对于按钮敲击，这个对象是按钮；对于鼠标进入事件，这个对象是所进入的窗口部件。

第二步，确定事件对象是否被允许处理事件。

事件处理过程检查的下一步是看相关窗口部件当前是否被允许去处理事件。 通过调用 `wx.EvtHandler` 的 `SetEvtHandlerEnabled(boolean)` 方法，一个窗口可以被设置为允许或不允许事件处理。不允许事件处理的结果是该窗口部件在事件处理中被完全绕过，与该对象关联的绑定对象也不会被搜索，并且在这步中的处理没有向下的分支。

在事件处理器级使一个窗口部件有效或无效与在用户界面级(`UI`) 不一样。在 `UI` 级使一个窗口部件无效或有效，使用 `wx.Window` 的方法 `Disable()` 和 `Enable()` 。在 `UI` 级使一个窗口部件无效意味用户不能与这个无效的窗口部件交互。通常无效的窗口部件在屏幕上以灰化的状态表示。一个在 `UI` 级无效的窗口不能产生任何事件；但是，如果它对于别的事件是容器的级别，它仍然能够处理它接受到的事件。本节的剩余内容，我们将在 `wx.EvtHandler` 层面上使用有效和无效，这涉及到窗口部件是否被允许处理事件。

对于初始对象有效或无效状态的检查，这发生在 `ProcessEvent()` 方法中，该方法由 `wxPython` 系统调用以开始和处理事件分配机制。我们将在事件处理过程中一再看到 `ProcessEvent()` 方法，它是类 `wx.EvtHandler` 中的方法，它实际上执行图3.3所描绘的大量事件处理。如果 `ProcessEvent()` 方法最后完成了事件处理，则 `ProcessEvent()` 返回 `True` 。如果一个处理器被发现和组合事件被处理，则认为处理完成。处理器函数可以通过调用 `wx.Event` 的 `Skip()` 方法来显式地请求进一步的处理。另处，如果初始对象是 `wx.Window` 的一个子类，那么它能够使用一个称为 `validator` 的对象来过滤事件。 `Validator` 将在第九章中详细讨论。

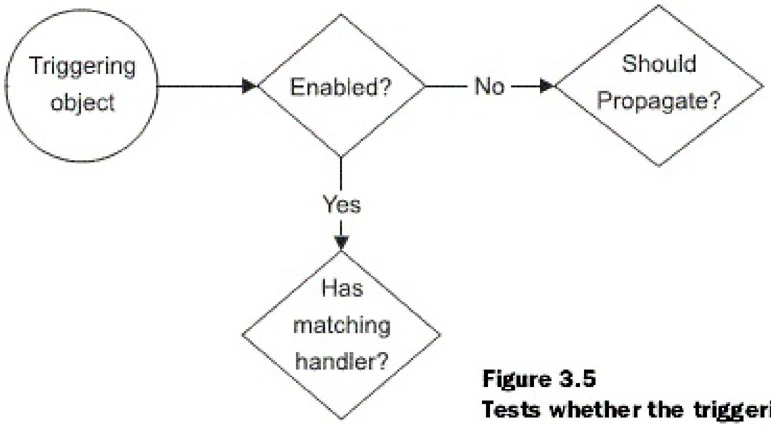


Figure 3.5
Tests whether the triggering object is enabled

第三步 定位绑定器对象

如图3.6所示

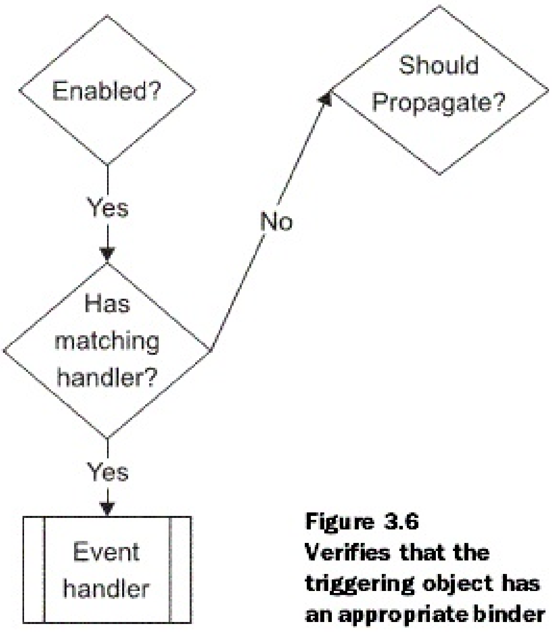


Figure 3.6
Verifies that the triggering object has an appropriate binder

然后 `ProcessEvent()` 方法寻找一个绑定器对象，该绑定器对象确定当前对象和事件类型之间的绑定。

如果对象自身的绑定器没有被找到，那么向上到该对象的超类中去寻找。如果一个绑定器对象被发现， `wxPython` 调用相关的处理器函数。在处理器被调用后，该事件的事件处理停止，除非处理器函数显式地要求作更多的处理。

在例子3.3中，因为在按钮对象，绑定器对象 `wx.EVT_ENTER_WINDOW` ，和相关的方法 `OnEnterWindow()` 之间定义了绑定，所以鼠标进入事件被捕获， `OnEnterWindow()` 方法被调用。由于我们没有绑定鼠标敲击事件 `wx.EVT_LEFT_DOWN` ，在这种情况下， `wxPython` 将继续搜索。

第四步 决定是否继续处理

如图3.7所示

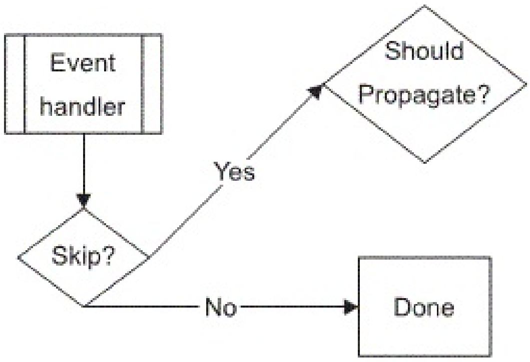


Figure 3.7 The event handler calls `Skip()`, and processing continues

在调用了第一个事件处理器之后，`wxPython` 查看是否有进一步的处理要求。事件处理器通过调用 `wx.Event` 的方法 `Skip()` 要求更多的处理。如果 `Skip()` 方法被调用，那么处理将继续，并且任何定义在超类中的处理器在这一步中被发现并执行。`Skip()` 方法在处理中的任一点或处理器所调用的任何代码中都可以被调用。`Skip()` 方法在事件实例中设置一个标记，在事件处理器方法完成后，`wxPython` 检查这个标记。在例3.3中，`OnButtonClick()` 不调用 `Skip()` ，因此在那种情况下，处理器方法结束后，事件处理完成。在另两个事件处理器中调用了 `Skip()` ，所以系统将保持搜索"匹配事件绑定"，最后对于原窗口部件的鼠标进入和离开事件调用默认的功能，如鼠标位于其上的事件。

第五步 决定是否展开 如图3.8所示

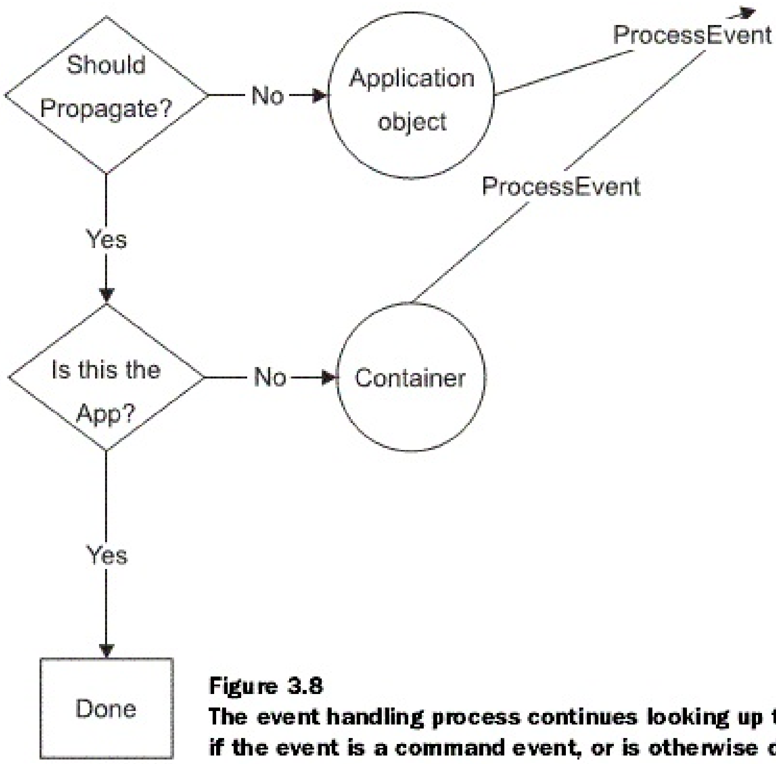


Figure 3.8
The event handling process continues looking up the container hierarchy if the event is a command event, or is otherwise declared to propagate

最后，`wxPython` 决定是否将事件处理向上展开到容器级以发现一个事件处理器。所谓的容器级是从一个特定的窗口部件到顶层框架的路径，这个路径是从窗口部件到它的父容器，一直向上沿升。

如果当前对象没有关于该事件的一个处理器，或如果处理器调用了 `Skip()` ， `wxPython` 将决定是否这个事件将沿容器级向上展开。如果决定不，那么在 `wx.App` 实例中再找寻一次处理器，然后停止。如果决定是，则事件处沿该窗口的容器级向上搜索，直到发现适当的绑定，或到达顶层框架对象，或到达一个 `wx.Dialog` 对象（即使这个对话框不是顶级的）。如果 `ProcessEvent()` 返回 `True` ，事件则被认为发现了一个适当的绑定，这表示处理完成。到达一个 `wx.Dialog` 停止的目的是防止父框架被来自对话框的无关的或未预期的假事件干扰。

一个事件是否向上展开至容器级，这是每个事件实例的一个动态属性，尽管实际上默认值几乎总是使用那几个。默认情况，只有 `wx.CommandEvent` 及其子类的实例向上展开至容器级。其它的所有事件不这样做。

在例3.3中，按钮敲击事件得到处理。在 `wx.Button` 上敲击鼠标产生一个命令类型的事件 `wx.EVT_BUTTON` 。由于 `wx.EVT_BUTTON` 属于一个 `wx.CommandEvent` ，所以 `wxPython` 在这个按钮对象中?已鞍蠖去O芎螅ä蔞咯蛏险箍偱寥菔骷叮è仁前磁サ母复翱趆。由于 `panel` 中没有相匹配的绑定，所以又向上至 `panel` 的父窗口 `frame` 。由于 `frame` 中有匹配的绑定，所以 `ProcessEvent()` 调用相关函数 `OnButtonClick()` 。

第五步同时也说明了为什么鼠标进入和离开事件必须被绑定到按钮而不是框架。由于鼠标事件不是 `wx.CommandEvent` 的子类，所以鼠标进入和离开事件不向上展开至容器级。如果鼠标进入和离开事件被绑定到了框架，那么当鼠标进入或离开框架时， `wxPython` 触发鼠标进入或离开事件。

在这种方式中，命令事件是被优先对待的。因为它们被认为是高级事件，表示用户正在应用程序空间中做一些事，而非窗口系统。窗口系统类型事件只对窗口部件感兴趣，而应用级事件对容器级。这个规则不防碍我们在任何地方声明绑定，不管被绑定的是什么对象或什么对象定义事件处理器。例如，即使这个绑定的鼠标敲击事件针对于按钮对象，而绑定则被定义在这个框架类中，且调用这个框架内的方法。换句话说，低级的非命令事件通常用于窗口部件或一些系统级的通知，如鼠标敲击、按键按下、绘画请求、调整大小或移动。另一方面，命令事件，如在按钮上敲击鼠标、或列表框上的选择，通常由窗口部件自己生成。例如，在适当的窗口部件上按下和释放鼠标后，按钮命令事件产生。

最后，如果遍历了容器级后，事件没有被处理，那么应用程序的 `wx.App` 对象调用 `ProcessEvent()` 。默认情况下，这什么也不做，但是你可以给你的 `wx.App` 增加事件绑定，以便以非标准的方式来传递事件。例如，假如你在写一个 `GUI` 构建器，你可能想把你构建器窗口中的事件传到你的代码窗口中，即使它们都是顶级窗口。方法之一是捕获应用程序对象中的事件，并把它们传递到代码窗口上。

使用Skip()方法

事件的第一个处理器函数被发现并执行完后，该事件处理将终止，除非在处理器返回之前调用了该事件的 `Skip()` 方法。调用 `Skip()` 方法允许另外被绑定的处理器被搜索，搜索依据3.4.1节中的第四步中声明的规则，因此父类和父窗口被搜索，

就如同这第一个处理器不存在一样。在某些情况下，你想继续处理事件，以便原窗口部件的默认行为和你定制的处理能被执行。例3.4显示了一个使用 `Skip()` 的例子，它使得程序能够同时响应同一按钮上的鼠标左按键按下和按钮敲击。

例3.4 同时响应鼠标按下和按钮敲击

```
#!/usr/bin/env python

import wx

class DoubleEventFrame(wx.Frame):

def __init__(self, parent, id):
wx.Frame.__init__(self, parent, id, 'Frame With Button',
size=(300, 100))
self.panel = wx.Panel(self, -1)
        self.button = wx.Button(self.panel, -1, "Click Me", pos=
(100, 15))
self.Bind(wx.EVT_BUTTON, self.OnButtonClick,
            self.button)    #1 绑定按钮敲击事件
        self.button.Bind(wx.EVT_LEFT_DOWN, self.OnMouseDown)   #
2 绑定鼠标左键按下事件

def OnButtonClick(self, event):
self.panel.SetBackgroundColour('Green')
        self.panel.Refresh()

def OnMouseDown(self, event):
self.button.SetLabel("Again!")
        event.Skip()   #3 确保继续处理

if __name__ == '__main__':
app = wx.PySimpleApp()
    frame = DoubleEventFrame(parent=None, id=-1)
frame.Show()
app.MainLoop()
```

#1 这行绑定按钮敲击事件到 `OnButtonClick()` 处理器，这个处理器改变框架的背景色。

#2 这行绑定鼠标左键按下事件到 `OnMouseDown()` 处理器，这个处理器改变按钮的标签文本。由于鼠标左键按下事件不是命令事件，所以它必须被绑定到按钮（ `self.button.Bind` ）而非框架（ `self.Bind` ）。

当用户在按钮上敲击鼠标时，通过直接与底层操作系统交互，鼠标左键按下事件首先被产生。通常情况下，鼠标左键按下事件改变按钮的状态，随着鼠标左键的释放，产生了 `wx.EVT_BUTTON` 敲击事件。由于行#3的 `Skip()` 语句， `DoubleEventFrame` 维持处理。没有 `Skip()` 语句，事件处理规则发现在#2创建的绑定，而在按钮能产生 `wx.EVT_BUTTON` 事件之前停止。由于 `Skip()` 的调用，事件处理照常继续，并且按钮敲击被创建。

记住，当绑定低级事件时如鼠标按下或释放，`wxPython` 期望捕获这些低级事件以便生成进一步的事件，为了进一步的事件处理，你必须调用 `Skip()` 方法，否则进一步的事件处理将被阻止。

在应用程序对象中还包含哪些其它的属性？

要更直接地管理主事件循环，你可以使用一些 `wx.App` 方法来修改它。例如，按你的计划，你可能想开始处理下一个有效的事件，而非等待 `wxPython` 去开始处理。如果你正在执行一个长时间的过程，并且不想图形界面被冻结，那么这个特性是必要的，通常你不需要使用这节中的这些方法，但是，这些性能有时是很重要的。

下表3.4列出了你可以用来修改主循环的 `wx.App` 方法：

`Dispatch()`：迫使事件队列中的下一个事件被发送。通过 `MainLoop()` 使用或使用在定制的事件循环中。 `Pending()`：如果在 `wxPython` 应用程序事件队列中有等待被处理的事件，则返回 `True` 。 `Yield(onlyIfNeeded = False)`：允许等候处理的 `wxWidgets` 事件在一个长时间的处理期间被分派，否则窗口系统将被锁定而不能显示或更新。如果等候处理的事件被处理了，则返回 `True`，否则返回 `False` 。 `onlyIfNeeded` 参数如果为 `True`，那么当前的处理将让位于等候处理的事件。如果该参数为 `False`，那么递归调用 `Yield` 是错误的。 这里也有一个全局函数 `wx.SafeYield()`，它阻止用户在 `Yield` 期间输入数据（这通过临时使用来输入的窗口部件无效来达到目的），以免干扰 `Yield` 任务。

另一管理事件的方法是通过定制的方式，它创建你自己的事件类型，以匹配你的应用程序中特定的数据和窗口部件。下一节我们将讨论如何创建你自己的定制事件。

如何创建自己的事件？

尽管这是一个更高级的主题，但是我们将在这里讨论定制事件。当你第一次阅读的时候，你可以跳过并且以后再回过头来读。 为了要与 `wxPython` 提供的事件类相区别，你可以创建你自己定制的事件。你可以定制事件以响应哪些针对你的应用程序的数据更新或其它改变，此处定制的事件必须负责你的自定义数据。创建定制的事件类的另一个原因是：你可以针对所定制的窗口部件，使用它自己独特的命令事件类型。下一节中，我们将看一个定制窗口部件的例子。

为一个定制的窗口部件定义一个定制的事件

。



Figure 3.9 The custom two-button widget. Clicking both buttons in succession triggers a change in the window title.

图3.9显示了这个窗口部件，一个画板(`panel)`包含了两个按钮。自定义的事件`TwoButtonEvent`仅当用户敲击了这两个按钮之后被触发。这个事件包含了一个关于用户在该部件上敲击次数的计数。

创建自定义事件的步骤：

1、定义一个新的事件类，它是`wxPython`的`wx.PyEvent`类的子类。如果你想这个事件被作为命令事件，你可以创建`wx.PyCommandEvent`的子类。像许多`wxPython`中的覆盖一样，一个类的`py`版本使得`wxWidget`系统明白用`Python`写的覆盖C++方法的方法。

2、创建一个事件类型和一个绑定器对象去绑定该事件到特定的对象。

3、添加能够建造这个新事件实例的代码，并且使用`ProcessEvent()`方法将这个实例引入事件处理系统。一旦该事件被创建，你就可以像使用其它的`wxPython`事件一样创建绑定和处理器方法。

下例3.5显示了管理窗口部件的代码：

```
import wx

class TwoButtonEvent(wx.PyCommandEvent):   #1 定义事件
def __init__(self, evtType, id):
        wx.PyCommandEvent.__init__(self, evtType, id)
self.clickCount = 0

def GetClickCount(self):
return self.clickCount

def SetClickCount(self, count):
self.clickCount = count

myEVT_TWO_BUTTON = wx.NewEventType() #2 创建一个事件类型
EVT_TWO_BUTTON = wx.PyEventBinder(myEVT_TWO_BUTTON, 1)  #3 创建一
个绑定器对象

class TwoButtonPanel(wx.Panel):
def __init__(self, parent, id=-1, leftText="Left",
rightText="Right"):
wx.Panel.__init__(self, parent, id)
self.leftButton = wx.Button(self, label=leftText)
self.rightButton = wx.Button(self, label=rightText,
pos=(100,0))
self.leftClick = False
self.rightClick = False
self.clickCount = 0
#4 下面两行绑定更低级的事件
self.leftButton.Bind(wx.EVT_LEFT_DOWN, self.OnLeftClick)
self.rightButton.Bind(wx.EVT_LEFT_DOWN, self.OnRightClick)

def OnLeftClick(self, event):
self.leftClick = True
```

```
self.OnClick()
        event.Skip()   #5 继续处理

def OnRightClick(self, event):
self.rightClick = True
self.OnClick()
        event.Skip()   #6 继续处理

def OnClick(self):
self.clickCount += 1
if self.leftClick and self.rightClick:
self.leftClick = False
self.rightClick = False
            evt = TwoButtonEvent(myEVT_TWO_BUTTON, self.GetId())
 #7 创建自定义事件
            evt.SetClickCount(self.clickCount)   # 添加数据到事件
            self.GetEventHandler().ProcessEvent(evt)  #8 处理事件

class CustomEventFrame(wx.Frame):
def __init__(self, parent, id):
wx.Frame.__init__(self, parent, id, 'Click Count: 0',
size=(300, 100))
panel = TwoButtonPanel(self)
        self.Bind(EVT_TWO_BUTTON, self.OnTwoClick, panel) #9 绑定
自定义事件

    def OnTwoClick(self, event):   #10 定义一个事件处理器函数
self.SetTitle("Click Count: %s" % event.GetClickCount())

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = CustomEventFrame(parent=None, id=-1)
frame.Show()
app.MainLoop()
```

说明：

#1 这个关于事件类的构造器声明为 `wx.PyCommandEvent` 的一个子类。`wx.PyEvent` 和 `wx.PyCommandEvent` 是 `wxPython` 特定的结构，你可以用来创建新的事件类并且可以把C++类和你的 `Python` 代码连接起来。如果你试图直接使用 `wx.Event` ，那么在事件处理期间 `wxPython` 不能明白你的子类的新方法，因为C++事件处理不了解该 `Python` 子类。如果你 `wx.PyEvent` ，一个对该 `Python` 实例的引用被保存，并且以后被直接传递给事件处理器，使得该 `Python` 代码能被使用。

#2 全局函数 `wx.NewEventType()` 的作用类似于 `wx.NewId()` ；它返回一个唯一的事件类型 `ID` 。这个唯一的值标识了一个应用于事件处理系统的事件类型。

#3 这个绑定器对象的创建使用了这个新事件类型作为一个参数。这第二个参数的取值位于[0,2]之间，它代表 `wxId` 标识号，该标识号用于 `wx.EvtHandler.Bind()` 方法去确定哪个对象是事件的源。

#4 为了创建这个新的更高级的命令事件，程序必需响应特定的用户事件，例如，在每个按钮对象上的鼠标左键按下。依据哪个按钮被敲击，该事件被绑定到 `OnLeftClick()` 和 `OnRightClick()` 方法。处理器设置了布尔值，以表明按键是否被敲击。

#5 #6 `Skip()` 的调用允许在该事件处理完成后的进一步处理。在这里，这个新的事件不需要 `skip` 调用；它在事件处理器完成之前被分派了(`self.OnClick()`)。但是所有的鼠标左键按下事件需要调用 `Skip()` ，以便处理器不把最后的按钮敲击挂起。这个程序没有处理按钮敲击事件，但是由于使用了 `Skip()` ， `wxPython` 在敲击期间使用按钮敲击事件来正确地绘制按钮。如果被挂起了，用户将不会得到来自按钮按下的反馈。

#7 如果两个按钮都被敲击了，该代码创建这个新事件的一个实例。事件类型和两个按钮的 `ID` 作为构造器的参数。通常，一个事件类可以有多个事件类型，尽管本例中不是这样。

#8 `ProcessEvent()` 的调用将这个新事件引入到事件处理系统中， `ProcessEvent()` 的说明见3.4.1节。 `GetEventHandler()` 调用返回 `wx.EvtHandler` 的一个实例。大多数情况下，返回的实例是窗口部件对象本身，但是如果其它的 `wx.EvtHandler()` 方法已经被压入了事件处理器堆栈，那么返回的将是堆栈项的项目。

#9 该自定义的事件的绑定如同其它事件一样，在这里使用#3所创建的绑定器。

#10 这个例子的事件处理器函数改变窗口的标题以显示敲击数。

至此，你的自定义的事件可以做任何预先存在的 `wxPython` 事件所能做的事，比如创建不同的窗口部件，它们响应同样的事件。创建事件是 `wxPython` 的定制的一个重要部分。

总结

1、 `wxPython` 应用程序使用基于事件的控制流。应用程序的大部分时间花费在一个主循环中，等待事件并分派它们到适当的处理器函数。

2、所有的 `wxPython` 事件是 `wx.Event` 类的子类。低级的事件，如鼠标敲击，被用来建立高级的事件，如按钮敲击或菜单项选择。这些由 `wxPython` 窗口部件引起的高级事件是类 `wx.CommandEvent` 的子类。大多的事件类通过一个事件类型字段被进一步分类，事件类型字段区分事件。

3、为了捕获事件和函数之间的关联， `wxPython` 使用类 `wx.PyEventBinder` 的实例。类 `wx.PyEventBinder` 有许多预定义的实例，每个都对应于一个特定的事件类型。每个 `wxPython` 窗口部件都是类 `wx.EvtHandler` 的子类。类 `wx.EvtHandler` 有一个方法 `Bind()` ，它通常在初始化时被调用，所带参数是一个事件绑定器实例和一个处理器函数。根据事件的类型，别的 `wxPython` 对象的 `ID` 可能也需要被传递给 `Bind()` 调用。

4、事件通常被发送给产生它们的对象，以搜索一个绑定对象，这个绑定对象绑定事件到一个处理器函数。如果事件是命令事件，这个事件沿容器级向上传递直到一个窗口部件被发现有一个针对该事件类型的处理器。一旦一个事件处理器被发现，对于该事件的处理就停止，除非这个处理器调用了该事件的 `Skip()` 方法。你可以允许多个处理器去响应一个事件，或去核查该事件的所有默认行为。主循环的某些方面可以使用 `wx.App` 的方法来控制。

5、在 `wxPython` 中可以创建自定义事件，并作为定制（自定义）的窗口部件的行为的一部分。自定义的事件是类 `wx.PyEvent` 的子类，自定义的命令事件是类 `wx.PyCommandEvent` 的子类。为了创建一个自定义事件，新的类必须被定义，并且关于每个事件类型（这些事件类型被这个新类所管理）的绑定器必须被创建。最后，这个事件必须在系统的某处被生成，这通过经由 `ProcessEvent()` 方法传递一个新的实例给事件处理器系统来实现。

在本章中，我们已经讨论了应用程序对象，它们对于你的 `wxPython` 应用程序是最重要的。在下一章，我们将给你看一个有用的工具，它是用 `wxPython` 写成的，它将帮助你使用 `wxPython` 进行开发工作。

第四章 用PyCrust使得wxPython更易处理

`PyCrust` 是一个图形化的 `shell` 程序，使用 `wxPython` 写成，它可以用来帮助你分析你的 `wxPython` 程序。

为何称它为 `PyCrust` ？这是因为当 `Patrick O’Brien` 使用 `wxPython` 创建一个交互式的 `Python` `shell` 时， `PyShell` 已被使用了，所以选用了 `PyCrust` 这个名字。

`PyCrust` 是 `Py` 包中的一部分， `Py` 包目前被包含在 `wxPython` 中。这个 `Py` 包还包含了其它相关功能的程序，这包括 `PyFilling` , `PyAlaMode` , `PyAlaCarte` , 和 `PyShell` 。这些程序每个都是想成为融图形化、点击环境、 `wxPython` 的交互、内省运行特点为一体。但是 `PyCrust` 表现最完善。

在这章中，我们将说明 `PyCrust` 和那些相关程序都干些什么，还有，你如何使用它们才能使得你用 `wxPython` 工作得更流畅。我们以谈论普通的 `Python` `shell` 作为开始，然后专门针对 `PyCrust` ，最后我们将涉及 `Py` 包中剩下的程序。

如何与wxPython程序交互？

与其它编程语言相比，`Python`的一个显著的特点是你可以以两种方式来使用它：你可以用它来运行存在的使用`Python`语言写的程序，或从命令提示符来交互地运行`Python`。交互地运行`Python`如同与`Python`解释器会话。

在下例4.1中，我们从命令行启动`Python`，并键入一些数学运算。`Python`启动后显示几行信息，然后是它的主提示符''。当你键入的东西要求额外的代码行时，`Python`显示它的次提示符'...'。

例4.1 简单的`Python`交互式会话

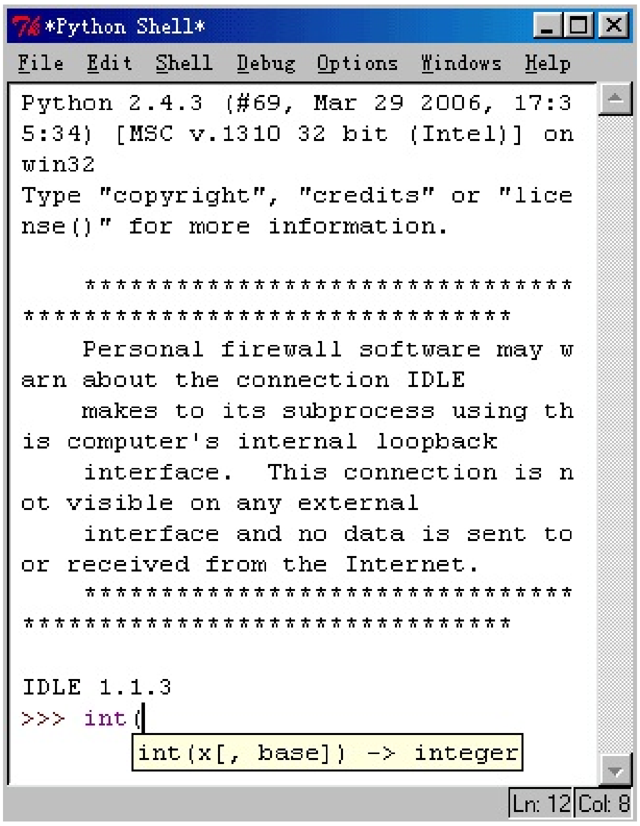
```
$ Python
Python 2.3.3 (#1, Jan 25 2004, 11:06:18)
[GCC 3.2.2 (Mandrake Linux 9.1 3.2.2-3mdk)] on linux2
Type "help", "copyright", "credits" o "license" for more informa
tion.
    2 + 2
4
    7 * 6
42
    5 ** 3
125
    for n in range(5):
...     print n * 9
...
0
9
18
27
36
```

交互式的`Python`不仅仅是一个好的桌面计算器，它也是一个好的学习工具，因为它提供了及时的反馈。当你有疑问时，你可以运行`Python`，键入几行试验性的代码，看`Python`如何反应，据此调整你的主要代码。学习`Python`或学习现有的`Python`代码是如何工作的，最好的方法之一就是交互式地调试。

`PyCrust`配置了标准的`Python` `shell`

当你交互式的使用`Python`工作时，你工作在一个称为`Python` `shell`的环境中，它类似于其它的`shell`环境，如微软平台的`DOS`窗口，或类`Unix`系统的`bash`命令行。

所有`Python` `shell`中最基本的是例4.1中所出现的，它是你从命令行启动`Python`时所见到的。虽然它是一个有用的`shell`，但是它基于文本的，而非图形化的，并且它不提供快捷方式或有帮助的提示。有几个图形化的`Python` `shell`已经被开发出来了，它们提供了这些额外的功能。最著名的是`IDLE`，它是`Python`发布版的标准部分。`IDLE`如下图4.1所示：



IDLE 看起来很像命令行 Python shell ，但是它有额外的特性如调用提示。

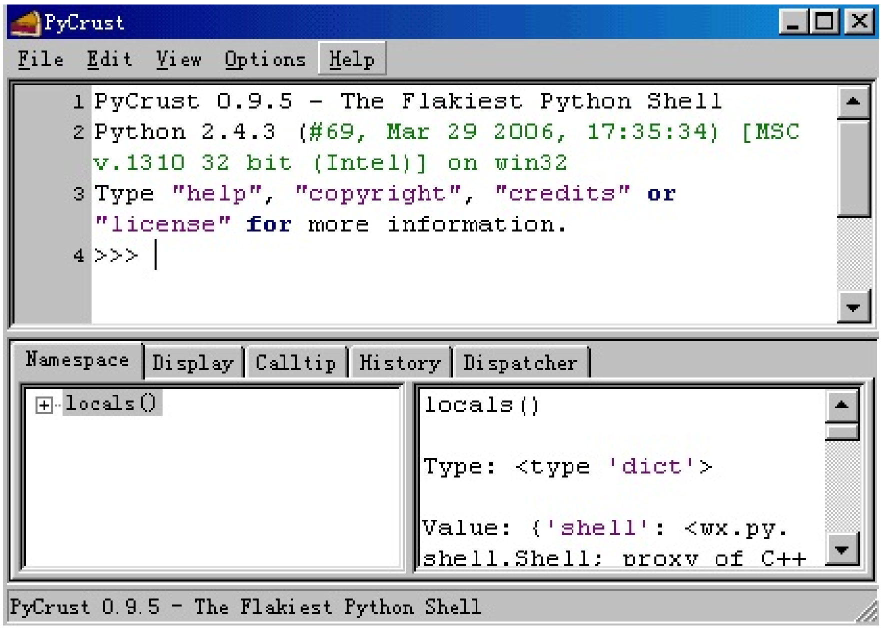
其它的 Python 开发工具，如 PythonWin 和 Boa Constructor ，包括了类似于 IDLE 中的图形化 Python shell 。虽然每种工具的 shell 各有一些有用的特性，如命令再调用(recall) 、自动完成、调用提示，但是没有一个工具完整包含了所有的特性 。在这种情况下， PyCrust 产生了， PyCrust 的目的之一就是提供所有现存的 Python shell 的特性。

创建 PyCrust 的另一个动机是：使用一个 GUI 工具包所写的代码不能工作在另一个不同的 GUI 工具包上。例如， IDLE 是用 Tkinter 写的，而不是 wxPython 。由于这样，如果你试图在 IDLE 的 Python shell 中引入和使用 wxPython 模块，那么你将陷入 wxPython 的事件循环与 Tkinter 事件循环间的冲突，结果将导致程序的冻结或崩溃。

事实上，这两个工具包将在控制事件循环上产生冲突。因此，如果你使用 wxPython 模块工作时想要内省运行特性，你的 Python shell 必须是用 wxPython 写的。由于没有现存的 Python shell 支持完整的特性， PyCrust 被创建来填补这种需要。

PyCrust的有用特性是什么？

现在，我们将关注 PyCrust 提供的 shell 的一些特性。 PyCrust 的 shell 看起来有些熟悉，这是因为它也显示了如同命令行 Python shell 相同的信息行和提示符。下图4.2显示了一个打开着的 PyCrust 的屏幕：



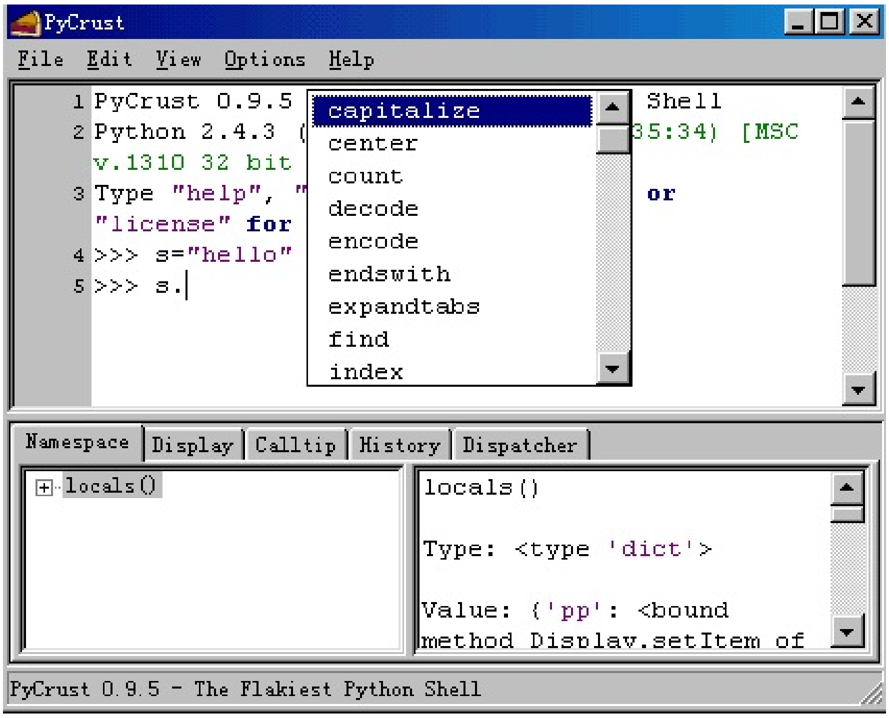
你应该注意一下这个 `PyCrust` 框架，它包含了一个 `wx.SplitterWindow` 控件，框架被分成两个区域：上部的区域看起来像通常的 `Python shell` ；底部的区域包含了一个 `Notebook` 控件，这个控件包含了不同的标签，默认标签显示的信息是有关当前的名字空间的。上部区域是 `PyCrust shell` ，它有几个有用的特性，我们将在下面几节讨论。

自动完成

当你在一个对象名后键入一点号时将引发自动完成功能。 `PyCrust` 将按字母顺序显示关于该对象的所有已知的属性的一个列表。当你在点号后输入字母时，在列表中的高亮选项将改变去匹配你所输入的字母。如果高亮选项正是你所要的，这时按下 `Tab` 键， `PyCrust` 将为你补全该属性名的其余部分。

在下图4.3中， `PyCrust` 显示一个字符串对象的属性的列表。这个自动完成的列表包含了该对象的所有属性和方法。

图4.3

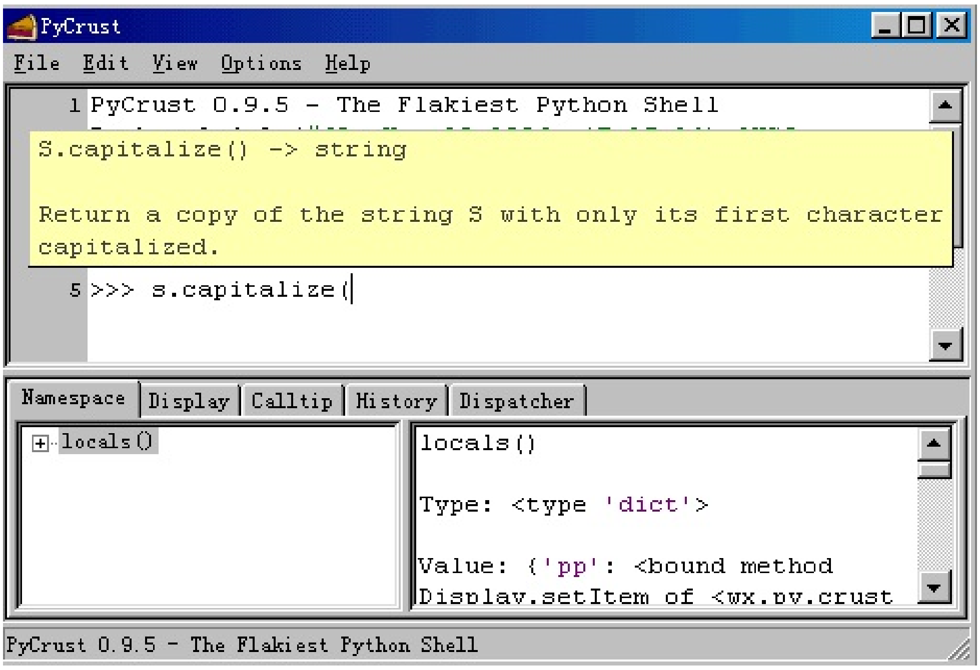


调用提示和参数默认

当你在一个可调用的对象名后键入左括号时， `PyCrust` 显示一个调用提示窗口（如图4.4），该窗口包含了所能提供的参数信息和文档字符串（如果可调用对象中定义了文档字符串的话）。

可调用对象可以是函数、方法、内建的或类。可调用对象的定义都可以有参数，并且可以有用来说明功能的文档字符串，以及返回值的类型。如果你知道如何使用该可调用对象，那么你可以忽略调用提示并继续键入。

图4.4



语法高亮

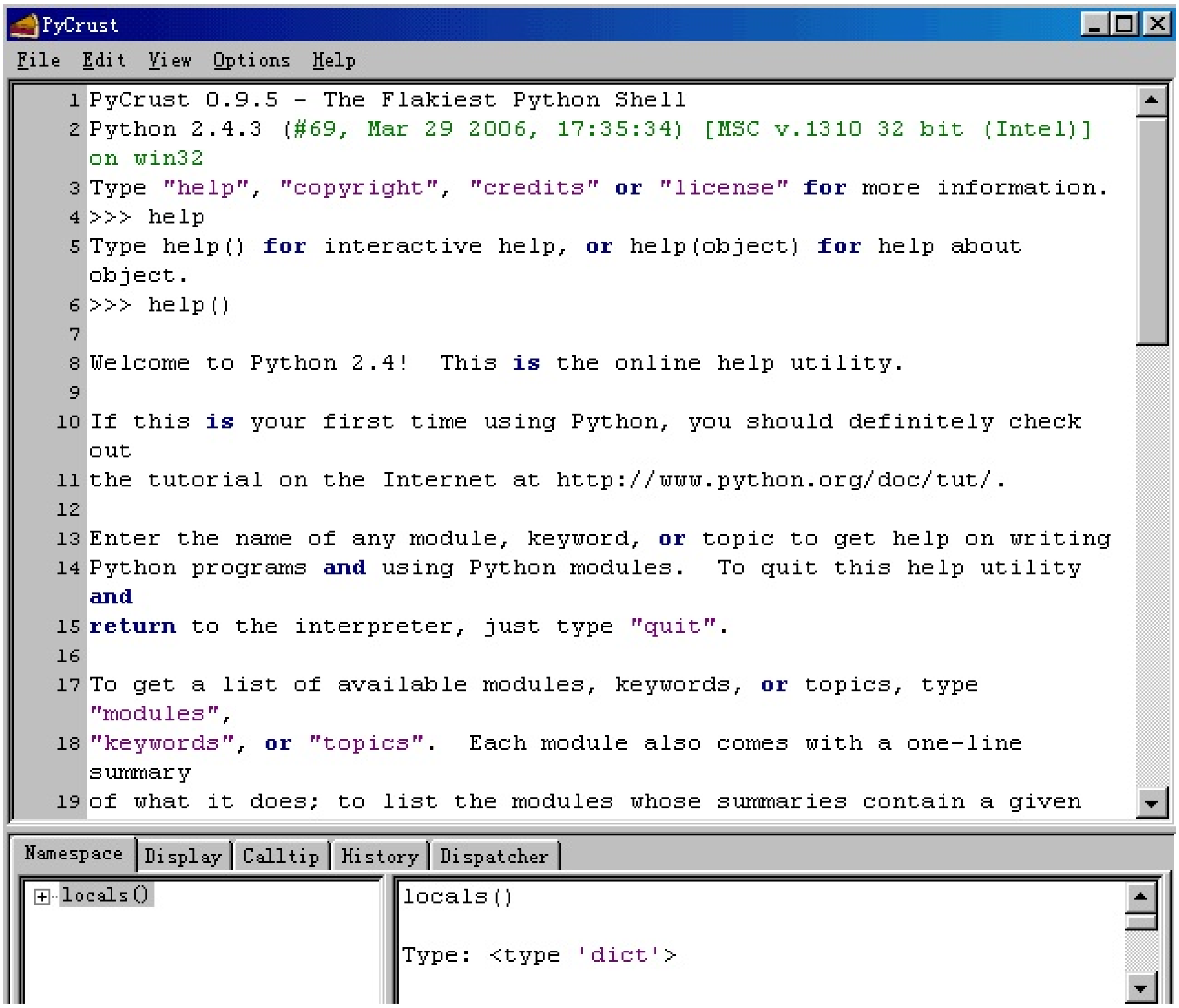
当你在 `shell` 中键入代码时， `PyCrust` 根据它的重要性改变文本的颜色。例如， `Python` 的关键词用一种颜色显示，原义字符串用另一种颜色，注释用另一种颜色。这就使得你可以通过颜色来确认你的输入是否有误。

Python

帮助

`PyCrust` 完整地提供了关于 `Python` 的帮助功能。 `Python` 的帮助功能显示了几乎所有 `Python` 方面的信息，如下图4.5所示

图4.5



`Python` 的帮助功能提供了另外一个提示符(`help)` 。在使用了 `help` 之后，你可以通过在 `help` 提示符之后键入 `quit` 来退出帮助模式，返回到通常的 `Python` 提示符( )。

命令重调用

在 `PyCrust` `shell` 中有多种方法可以用来减少重复输入。它们大都通过捕获你先前的键入来实现，如果有必要，你可以修改所捕获的内容，之后它们将之发送给 `Python` 解释器。

例如， `PyCrust` 维护着当前会话中你所键入的所有命令的一个历史记录。你可以从命令历史记录中重调用你先前键入的任何 `Python` 命令（一行或多行）。下表4.1显示了一个关于该功能的快捷键列表。

`Ctrl` +上箭头：获取前一个历史项 `Alt` +P：获取前一个历史项 `Ctrl` +下箭头：获取下一个历史项 `Alt` +N：获取下一个历史项 `Shift` +上箭头：插入前一个历史项 `Shift` +下箭头：插入下一个历史项 `F8` ：历史项命令补全（键入先前命令的少量字符并按 `F8` ） `Ctrl` + `Enter` ：在多行命令中插入新行

正如你所看到的，这儿有不同的命令用于获取和插入旧命令，它们通过 `PyCrust` 如何处理当前 `wxPythob` 提示符中所键入的文本被区分。要替换你的键入或插入一个旧的命令，可以使用快捷键来获取或插入一个历史项。

插入一行到一个多行命令中的工作与插入到一单行命令不同。要插入一行到一个多行命令，你不能只按 `Enter` 键，因为这样将把当前的命令发送给 `Python` 解释器。替代的方法是，按下 `Ctrl` + `Enter` 来插入一个中断到当前行。如果你处于行尾，那么一个空行被插入当前行之后。这个过程类似于你在一个通常的文本编辑中剪切和粘帖文本的方法。

最后一种重调用命令的方法是简单地将光标移到想要使用的命令，然后按 `Enter` 键。 `PyCrust` 复制该命令到当前的 `Python` 提示符。然后你可以修改该命令或按 `Enter` 键以将该命令提交给解释器。

快捷键让你可以快速地开发代码，并做每步的测试。例如，你可以定义一个新的 `Python` 类，创建该类的一个实例，并看它的行为如何。然后，你可以返回到这个类的定义，增加更多的方法或编辑已有的方法，并创建一个新的实例。通过这样的反复，你可以将你的类的定义做得足够好，然后将它粘帖到你的源代码中。

剪切和粘贴

你可能想重用在 `shell` 中已开发的代码，而避免重新键入。有时，你可能找到一些样例代码（可能来自在线的教程），你想把它用到一个 `Python` `shell` 中。 `PyCrust` 提供了一些简单的剪切和粘贴选项，列表于下表4.2

`Ctrl` +C：复制所选的文本，去掉提示符 `Ctrl` + `Shift` +C：复制所选的文本，保留提示符 `Ctrl` +X：剪切所选的文本 `Ctrl` +V：粘贴自剪贴板 `Ctrl` + `Shift` +V：粘贴自剪贴板的多个命令并运行

粘贴的另一个特性是： `PyCrust` 从所粘贴到 `PyCrust` `shell` 中的代码中识别并自动去掉标准的 `Python` 提示符。这使得复制教程或 `email` 信息中的例子代码，把它粘贴到 `PyCrust` 中，并测试它变得简单了，省去了手工的清理。

某些时候，当你复制代码时，你可能想去除 `PyCrust` 提示符，如当你复制代码到你的源文件中时。另一些时候，你可能想保留这个提示符，如录你复制例子到一个文档中，或把它发送到一个新闻组。当从 `shell` 复制时， `PyCrust` 对这两种情

况都提供了支持。

标准shell环境

在 `wxPython` 环境中， `PyCrust` 的行为尽可能地与命令行的 `Python shell` 相同。不同的是，一旦 `Python` 代码被输入到了 `PyCrust shell` 中，就没有办法来中断该代码的运行。例如，假定你在 `PyCrust` 中写了一个无限循环，如下所示：

- `while True:`

... `print` " `Hello` " ...

在你按下 `Enter` 之后，上面的代码被传送到 `Python` 解释器， `PyCrust` 停止响应。要中断这个无限的循环，必须关闭 `PyCrust` 程序。这个缺点是与命令行的 `Python shell` 对比而言的。命令行的 `Python shell` 保留了处理键盘中断(`Ctrl` +C）的能力。在命令行的 `Python shell` 中你会看到如下的行为：

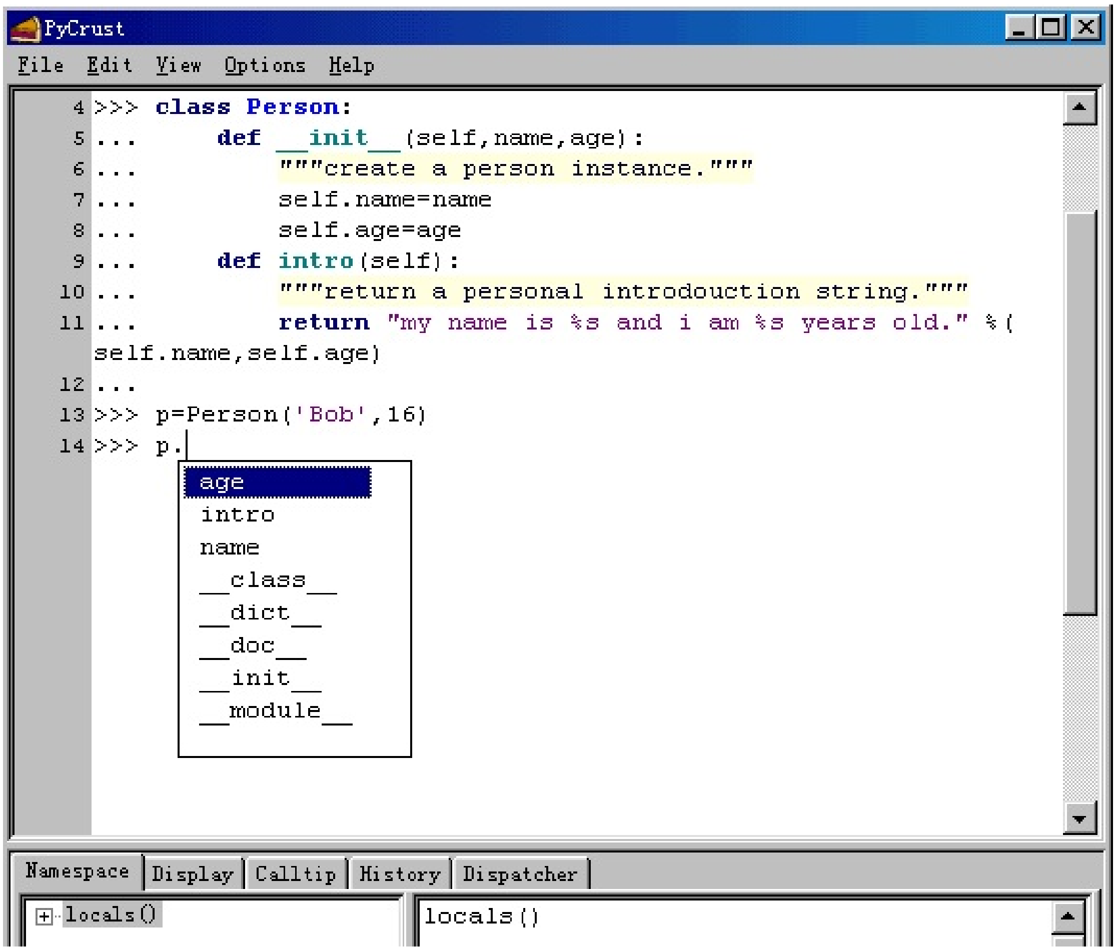
```
    while True:
...     print "Hello"
...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Traceback (most recent call last):
  File " stdin ", line 2, in ?
KeyboardInterrupt
```

在 `GUI` 环境中的事件处理的本质，使得设计出能够让 `PyCrust` 中断一个无限循环或在 `shell` 提示符中键入的长时间运行的代码序列的方法有很大的不同。将来的 `PyCrust` 版本可能会提供对这个缺点的一个解决办法。幸运的是，在 `PyCrust` 和标准命令 `shell` 之间只有这一个不同点。在其它方面， `PyCrust shell` 和命令行的 `Python shell` 工作的完全一样。

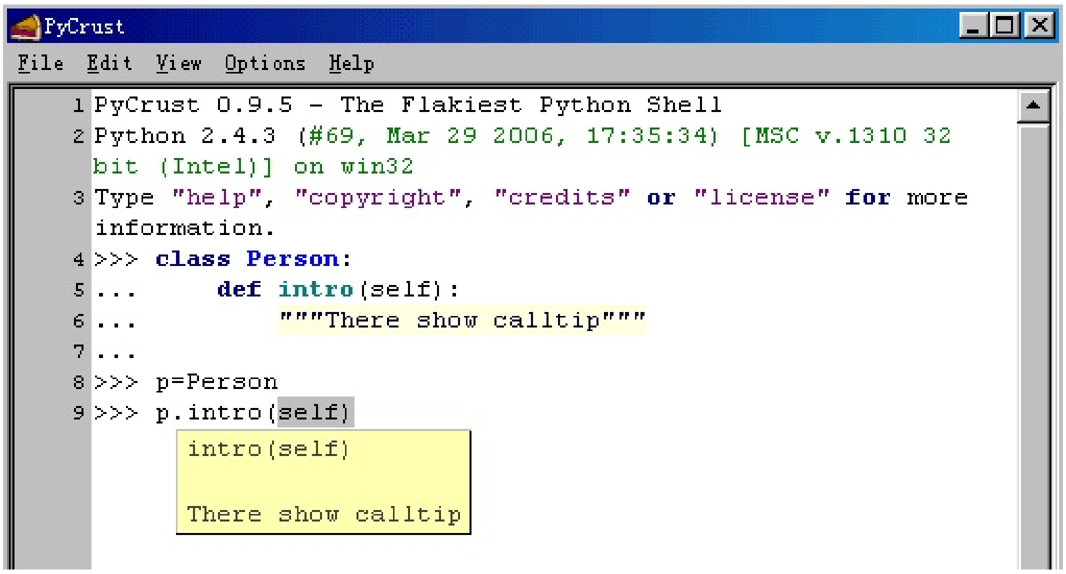
动态更新

当你在运行 `PyCrust` 时， `PyCrust` 的 `shell` 的所有特性都是动态地被更新的，这意味着，诸如“自动完成”和“调用提示”等特性是有效的，即使是在 `shell` 提示符中定义的对象。例如图4.6和4.7所显示的会话，那么我们定义并使用了一个类。

在图4.6中， `PyCrust` 为新类显示了自动完成选项。 图4.6



在图4.7中， `PyCrust` 显示了关于类所定义的新的方法的调用提示。 图4.7



PyCrust

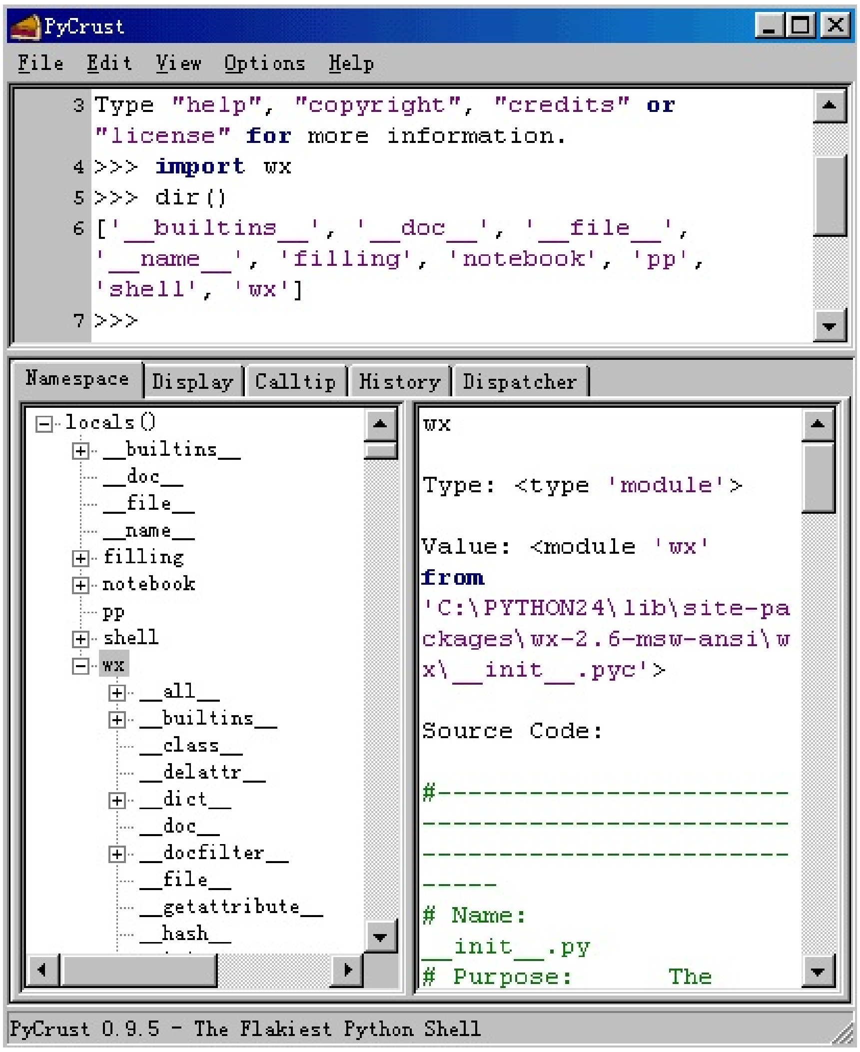
`notebook` 的标签是干什么的？

PyCrust 界面的下半部是一个 notebook 控件，notebook 控件包括了几个带有有用信息的标签。PyCrust 开始时，你所看到的标签是“ Namespace ”标签。

Namespace标签

如图4.8所示，Namespace 标签又被用 wx.SplitterWindow 控件分成两部分。左边包含一个树控件，它显示当前的名字空间，而右边显示在名字空间树中当前被选择的对象的细节。

图4.8



名字空间树呈现一个关于在当前名字空间中所有对象的层次关系的视图。如果你运行 Python 的内建函数 locals() ，这些对象将作为返回结果。在图4.8中，我们已经导入了 wx 包并在名字空间树中选择了它。右边显示了所选择的项目的名字，它的类型和它的当前值。如果对象有与之相关联的源代码， PyCrust 也将显示出来。这里， wx 是一个 wxPython 包，所以 PyCrust 显示 __init__.py 文件的源代码，该文件位于 wx 目录中。

右边显示的第一行是左边所选择的对象的全名，你可以把它复制并粘贴到 PyCrust shell 或你的应用程序源码中。例如，我们在 PyCrust 中引入 locale 模块并选择名字空间树中 locale / encoding_alias /' en '项，右边

就显示了所选对象的完整名，你可以把它复制并粘贴到 `PyCrust shell` 中，如下所示：

```
import locale
locale.encoding_alias['en']
'ISO8859-1'
```

这里， `PyCrust` 给我们提供了一个全名（ `locale.encoding_alias` [' `en` ']），它使用 `Python` 的索引（[' `en` ']）来引用 `encoding_alias` 目录中的指定项目。这个机制同样适用于列表(`list)` 。如果你在名字空间树中发现了你想用在你的代码中的东西，那么 `PyCrust` 给了你这精确语法去完成这个任务。

Display标签

- `Display` 标签中用于显示一个对象。 `PyCrust` 有一个内建函数 `pp()` ，这个函数使用 `Python` 的 `pprint` 模块为显示一个对象。使用中不需要显式地引入和重复使用 `pprint` ，在 `Display` 中，这些信息随对象的更新而每次更新。

例如，如果我们在 `PyCrust shell` 中有一个列表，我们要在 `Display` 标签中显示它的内容，我们可以在 `PyCrust shell` 中使用 `pp()` ，然后列表的内容就显示在 `Display` 标签中了。以后每当我们改变了列表的内容， `Display` 标签中的内容随即改变。

`Calltip` 标签显示了在 `Python shell` 中最近调用提示的内容。如果你的调用要求大量的参数，那么你可以选择 `Calltip` 标签。当使用 `wxPython` 包时，存在着大量的类，这些类有许多方法，这些方法又要求许多参数。例如，为了创建一人 `wx.Button` ，你可能要提供八个参数，有一个是必须提供的，其它七个有默认的值。 `Calltip` 标签显示了关于 `wx.Button` 构造器的细节，如下所示：

```
__init__(self, Window parent, int id=-1, String label=EmptyStrin
g,
Point pos=DefaultPosition, Size size=DefaultSize,
long style=0, Validator validator=DefaultValidator,
    String name=ButtonNameStr) -  Button

Create and show a button. The preferred way to create standard b
uttons
is to use a standard ID and an empty label. In this case wxWiget
s will
automatically use a stock label that corresponds to the ID given
. In
addition, the button will be decorated with stock icons under GT
K+2.
```

由于 wxPython 的类实际上是封装的C++的类，所以调用提示信息完全基于类的文档字符串。它们显示了底层C++类所需要的参数和类型信息。对于完全用 Python 语言定义的对象， PyCrust 检查它们以确定它的参数特性。

Session标签

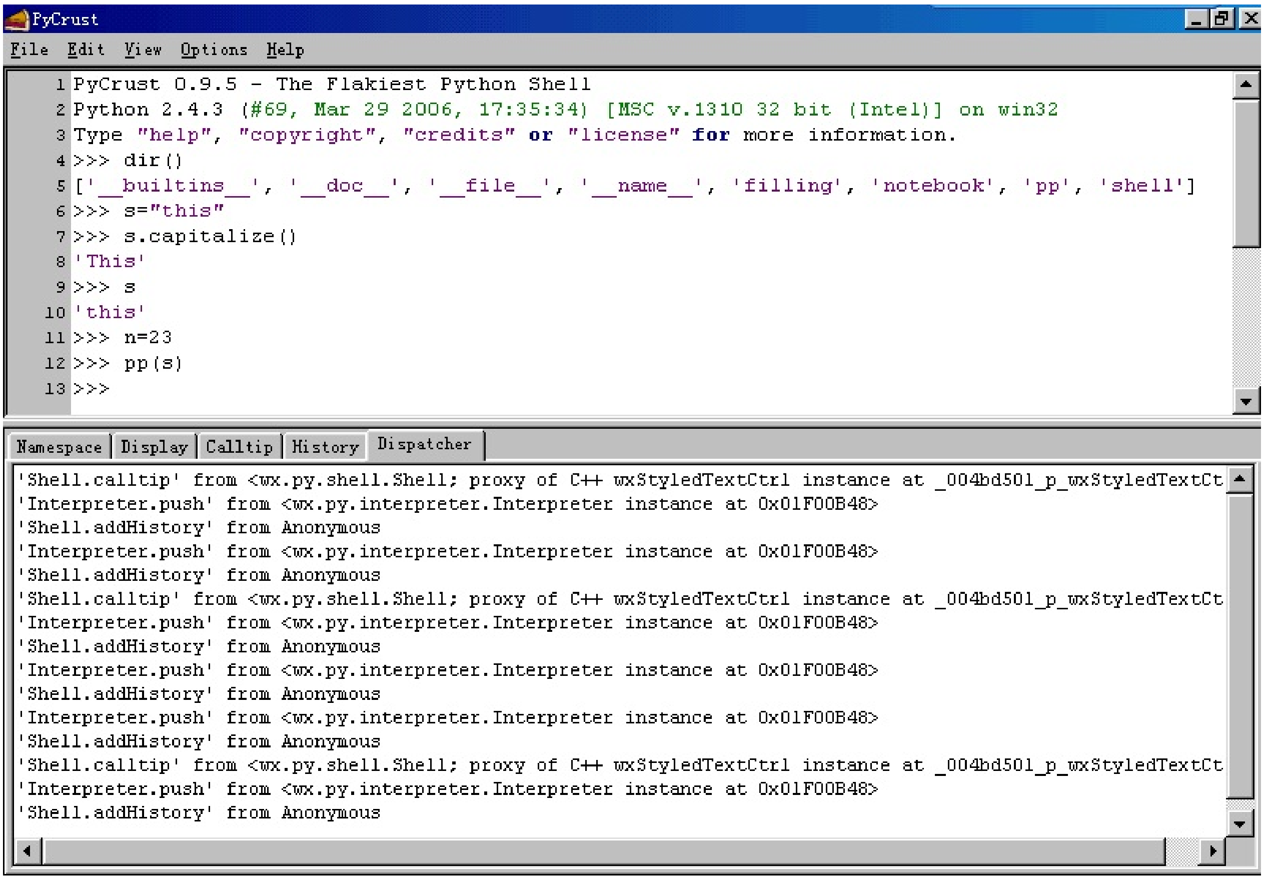
- Session 标签是一个简单的文本控件，它列出了在当前 shell 会话中所键入的所有命令。这使得剪切和粘贴命令以用在别处更为简单。

Dispatcher

标签

PyCrust 包括了一个名为 dispatcher 的模块，它提供了在一个应用程序中联系对象的机制。 PyCrust 使用 dispatcher 来维持它的界面的更新，主要是在命令从 shell 传送到 Python 解释器时。图4.9中的 Dispatcher 标签列出了关于信号经过分配机制后的路由。当使用 PyCrust 工作时，这是它的主要用处。

图4.9



这里的 Dispatcher 标签也演示了如何增加另一个标签到一个 wx.Notebook 控件。下面这个在 Dispatcher 标签上的文本控件的源码，演示了如何使用 dispatcher 模块：

```
class DispatcherListing(wx.TextCtrl):
    """Text control containing all dispatches for session."""

def __init__(self, parent=None, id=-1):
style = (wx.TE_MULTILINE | wx.TE_READONLY |
wx.TE_RICH2 | wx.TE_DONTWRAP)
wx.TextCtrl.__init__(self, parent, id, style=style)
dispatcher.connect(receiver=self.spy)

def spy(self, signal, sender):
"""Receiver for Any signal from Any sender."""
text = '%r from %s' % (signal, sender)
self.SetInsertionPointEnd()
start, end = self.GetSelection()
if start != end:
self.SetSelection(0, 0)
        self.AppendText(text + '\n')
```

现在我们已经看到了`PyCrust`作为独立的`Python` `shell`和名子空间检查器能够做些什么，下面让我们关注在你的`wxPython`程序中，`PyCrust`的其它一些用法。

如何将PyCrust应用于wxPython应用程序

。

让我们假设你已经用`wxPython`创建了一个程序，并且你的程序正在工作，现在你想更好地了解它是如何工作的。在这章的前面你已经看到了`PyCrust`的特性，它们看起来对于理解你的程序的功能是非常有用的。

通过将你的程序的名字传递给`PyWrap`，你能够用`PyCrust` `shell`来启动你的程序，不需要对你的程序作任何的改变。下例4.2显示了一个名为`spare.py`的程序，我们准备对它使用`PyCrust`。

例4.2

```
#!/usr/bin/env python

"""Spare.py is a starting point for simple wxPython programs."""

import wx

class Frame(wx.Frame):
    pass

class App(wx.App):

    def OnInit(self):
        self.frame = Frame(parent=None, id=-1, title='Spare')
        self.frame.Show()
        self.SetTopWindow(self.frame)
        return True

if __name__ == '__main__':
    app = App()
    app.MainLoop()
```

为了运行这个程序时使用 `PyCrust` ，要将该程序的全路径传递给 `PyWrap` 。在 `Linux` 上，命令行类似如下：

```
 $ pywrap spare.py
```

在 `windows` 下，命令行类似如下：

```
    F:\ python pywrap.py spare.py
```

在开始的时候， `PyWrap` 试图导入命令行所包括的模块。然后 `PyWrap` 在模块中寻找 `wx.App` 的子类，并创建子类的一个实例。之后， `PyWrap` 创建一个带有 `shell` 的 `wx.py.crust.CrustFrame` 窗口，把这个应用程序对象显示在 `PyCrust` 的名字空间树中，并且启动 `wxPython` 事件循环。

`PyWrap` 的源码显示在例子4.3中。它显示了如何用少量的代码将大量的功能增加到你的程序中。

例4.3

```
"""PyWrap is a command line utility that runs a python
program with additional runtime tools, such as PyCrust."""

__author__ = "Patrick K. O'Brien  pobrien@orbtech.com "
__cvsid__ = "$Id: PyCrust.txt,v 1.15 2005/03/29 23:39:27 robind
Exp $"
__revision__ = "$Revision: 1.15 $"[11:-2]

import os
import sys
import wx
from wx.py.crust import CrustFrame

def wrap(app):
    wx.InitAllImageHandlers()
    frame = CrustFrame()
    frame.SetSize((750, 525))
    frame.Show(True)
    frame.shell.interp.locals['app'] = app
    app.MainLoop()

def main(modulename=None):
    sys.path.insert(0, os.curdir)
    if not modulename:
        if len(sys.argv)   2:
            print "Please specify a module name."
            raise SystemExit
        modulename = sys.argv[1]
        if modulename.endswith('.py'):
            modulename = modulename[:-3]
    module = __import__(modulename)
    # Find the App class.
    App = None
    d = module.__dict__
    for item in d.keys():
        try:
            if issubclass(d[item], wx.App):
                App = d[item]
        except (NameError, TypeError):
            pass
    if App is None:
        print "No App class was found."
        raise SystemExit
    app = App()
    wrap(app)
if __name__ == '__main__':
    main()
```

运行了 `PyWrap` 命令之后，来自 `spare` 的简单的框架(`frame`) 和 `PyCrust` 的框架都显示出来。

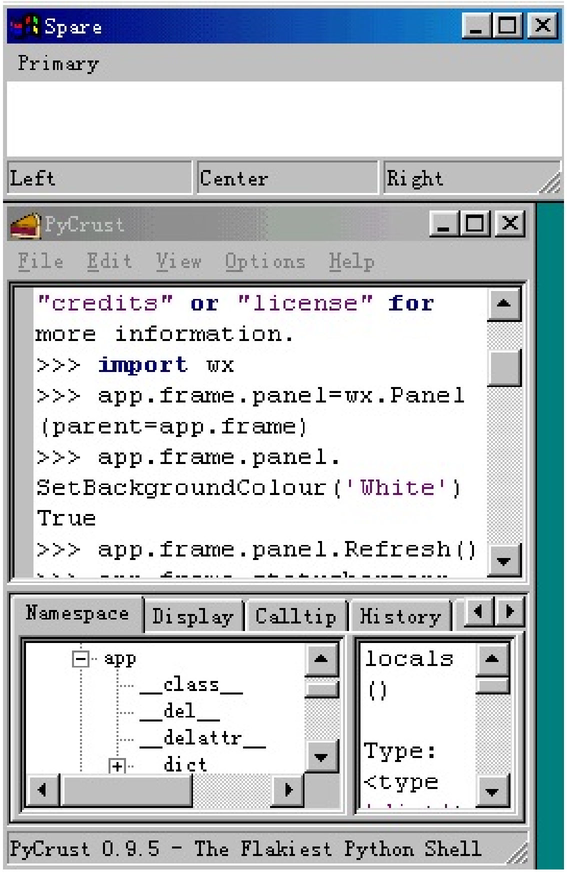
PyCrust in action

现在让我们看看，在 PyCrust shell 中我们对 spare.py 应用程序框架做些什么。图4.10显示了这个结果。我们将通过导入 wx 和增加一个画板到我们的框架作为开始：

```
import wx
app.frame.panel = wx.Panel(parent=app.frame)
app.frame.panel.SetBackgroundColour('White')
```

`True`

图4.10



增加到框架的画板开始时是默认的银灰色，然后它被改变到白色。然而，设置画板背景色不立即改变它的显示。这需要去触发一个事件来导致画板重绘，以使用它的新颜色属性。一个触发这样事件的方法是要求画板刷新自身：

```
app.frame.panel.Refresh()
```

现在一个白色的画板显示了，我们对于理解 wxPython 如何工作的细节又进了一步。

接下来，让我们增加一个状态栏：

```
app.frame.statusbar = app.frame.CreateStatusBar(number=3)
app.frame.statusbar.SetStatusText("Left", 0)
app.frame.statusbar.SetStatusText("Center", 1)
app.frame.statusbar.SetStatusText("Right", 2)
```

注意在不改变这个框架的尺寸情况下，这个状态栏在这个框架中是如何显示的。也要注意添加到三个状态栏中的文本的立即显示了出来，而不要求刷新。现在让我们增加一个菜单和一个菜单栏：

```
app.frame.menubar = wx.MenuBar()
menu = wx.Menu()
app.frame.menubar.Append(menu, "Primary")

app.frame.SetMenuBar(app.frame.menubar)
menu.Append(wx.NewId(), "One", "First menu item")

menu.Append(wx.NewId(), "Two", "Second menu item")
```

当你在 `PyCrust` `shell` 中处理你自己的 `wxPython` 对象时，注意改变对你正在运行的程序的影响。试试回答后面的问题。在框架中菜单何时才实际显示出来的？在程序运行的时候，你能改变菜单的哪些属性？你能够让它们无效吗？交互地探究这些可以帮助你更好的理解 `wxPython` ，同时当你写真实的代码时给你带来更大的自信。

到目前，我们已经花了很多节讨论 `PyCrust` ，我们下面准备看一看 `Py` 包的其余的东西。

在Py包中还有其它什么？

所有 `PyCrust` 中的程序都利用了 `Py` 包中的 `Python` 模块，诸如 `shell.py` , `crust.py` , `introspect.py` 和 `interpreter.py` 。这些程序是用来做 `PyCrust` 的建造块，你可以分别或一起使用它们。

`PyCrust` 代表了组装包含在 `Py` 包中功能模块的一各方法。 `PyShell` 是另一方法， `PyAlaMode` 是第三种。在这 些方法中，它们的底层代码大多数是相同的，只是外包装有所变化而已。因此，你可以把 `Py` 当做一个模块 库，你可以随意地在你的程序中的任何地方组装其中的模块，用来显示一个 `wxPython` `shell` 、一个代码编 辑器或运行时内省信息。

在 `Py` 包中，提供给用户界面功能的模块和没有这功能的模块有明显的区别。这个区别使得在你的程序中很 容易使用这些模块。以 `Py` 开头的模块是终端用户 `GUI` 程序，如 `PyCrust` , `PyShell` , `PyAlaMode` 和 `PyAlaCarte` 。在你的程序中，你不会想导入这些模块。下节说明终端用户模块。

使用GUI程序工作

下表4.3说明了用户级程序。

`PyAlaCarte`：简单的源代码编辑器。一次编辑一个文件。`PyAlaMode`：多文件源代码编辑器。每个文件分别显示在一个`notebook`标签中。第一个标签包含一个`PyCrust`分隔窗口。`PyCrust`：合并了`wxPython` `shell`和`notebook`标签，`notebook`包含一个名字空间树查看器。`PyFilling`：简单的名字空间树查看器。这个程序自己不是很有用。它的存在只是作为如何使用底层库的一个例子。`PyShell`：简单的`wxPython` `shell`界面，没有`PyCrust`中的`notebook`。功能上，`PyShell`中的`wxPython` `shell`和`PyCrust`中的是一样的。`PyWrap`：命令行工具，用以运行一个存在的程序和`PyCrust`框架，让你能够在`PyCrust` `shell`中处理这个应用程序。

使用支持模块工作

支持模块为终端用户提供了基本的功能，可以被导入你的程序中。这些模块是用来创建用户级`Py`程序的建

造块。下表4.4列出了这些支持模块，它们是`Py`包的一部分，说明如下：

`buffer`：支持文件编辑。`crust`：包含`PyCrust`应用程序独有的`GUI`元素。`dispatcher`：提供全局信号分派服务。`document`：`document`模块包含一个非常简单的`Document`类，这个类是一个小的文件类。`document`跟踪不同的文件属性，如名字和路径，并提供`read()`和`write()`方法。`Buffer`类委托这些低级的读写操作给一个`Document`实例。`editor`：包含所有显示在`PyAlaCarte`和`PyAlaMode`程序中的`GUI`编辑部分。`editwindow`：这个`editwindow`模块包含了一个简单的`EditWindow`类。这个类继承自`wx.stc.StyledTextCtrl`(`STC)`，并且提供了`Py`包中的`STC`的三种主要用法的所有共同的特性，这三种主要用法是：作为一个`Python` `shell`,作为一个源代码编辑器和作为一个只读源码显示器。`filling`：包含所有的`GUI`控件，这些`GUI`控件让用户能够浏览对象名字空间并显示那些对象运行时的信息。`frame`：`frame`模块定义了一个`Frame`类，这个`Frame`类是`Py`包中所有其它`frames`的基类。菜单项根据当前状态和上下文不断地自我更新。`images`：`images`模块包含被不同`Py`程序使用的图标。`interpreter`：`Interpreter`类负责提供自动完成列表，调用提示信息等。`introspect`：为一些方面提供多种内省类型，像调用提示和命令自动完成。`pseudo`：这个模块定义文件类类，文件类允许`Interpreter`类去重定向`stdin`,`stdout`,`stderr`。`shell`：这个模块包含`GUI`元素，这些`GUI`元素定义显示在`PyCrust`,`PyShell`和`PyAlaMode`中的`Python` `shell`的界面。`version`：这个模块包含一个名为`VERSION`的字符串变量，`VERSION`代表`Py`当前的版本。

下面我们讨论更复杂的模块。

`buffer` 模块

`buffer` 模块包含一个 `Buffer` 类，这个类支持文件的通常编辑。 `buffer` 有一些方法，例如 `new()` , `open()` , `hasChanged()` , `save()` ,和 `saveAs()` 。文件操作基于 `buffer` 所委托的 `Document` 类的实例， `Document` 类定义在 `document` 模块中。文件内容的实际编辑是通过 `Editor` 类的一个或多个实例发生的， `Editor` 类定义在 `editor` 模块中。 `buffer` 扮演一个在一个或多个编辑器和实际物理文件之间的中间人。

`Buffer` 类的一个独特的手法是每个 `buffer` 实例都分配了它自己的 `Python` 解释器实例。这个特点使得 `buffer` 能够被用在那些当编辑 `Python` 源代码文件时需要提供自动完成，调用提示和其它运行时帮助的应用程序中 。每个 `buffer` 解释器都是完全独立的，并且在 `buffer` 的 `updateNamespace()` 方法被调用时更新。下例4.4显示了 `updateNamespace()` 方法的源代码。

例4.4

```
def updateNamespace(self):
        """Update the namespace for autocompletion and calltips.
 Return True #if updated, False if there was an error."""

if not self.interp o not hasattr(self.editor, 'getText'):
        return False
        syspath = sys.path
        sys.path = self.syspath
        text = self.editor.getText()
        text = text.replace('\r\n', '\n')
        text = text.replace('\r', '\n')
        name = self.modulename o self.name
        module = imp.new_module(name)
        newspace = module.__dict__.copy()
        try:
                try:
                        code = compile(text, name, 'exec')
                except:
                        raise
                try:
                        exec code in newspace
                except:
                        raise
                else:
                        # No problems, so update the namespace.
                        self.interp.locals.clear()
                        self.interp.locals.update(newspace)
                        return True
        finally:
                sys.path = syspath
                for m in sys.modules.keys():
                if m not in self.modules:
                        del sys.modules[m]
```

这个方法使用 `Python` 内建的 `compile` 方法编译编辑器中的文本，然后使用关键词 `exec` 来执行。如果编译成 功，将放置若干变量到 `newspace` 名字空间中。通过用执行的结果重置解释器的局部名字空间，解释器支持 访问定义在编辑器的 `buffer` 中的任何类，方法或变量。

`crust` 模块

`crust` 模块包含6个 `GUI` 元素，它们专门用于 `PyCrust` 应用程序的。这最常用的类是 `CrustFrame` ，它是 `wx.Frame` 的子类。如果你再看一下例4.3，你能看到 `PyWrap` 程序是如何导入 `CrustFrame` 并创建其一个实例的。这是嵌入一个 `PyCrust` 框架到你自己的程序中的最简单的方法。如果你想要比一个完整的框架更小的东西，你可以使用下表4.5所列的一个或多个类。

表4.5

`Crust` ：基于 `wx.SplitterWindow` 并包含一个 `shell` 和带有运行时信息的 `notebook` 。 `Display` ：样式文本控件，使用 `Pretty` `Print` 显示一个对象。 `Calltip` ：文本控件，包含最近 `shell` 调用帮助。 `SessionListing` ：文本控件，包含关于一个会话的所有命令。 `DispatcherListing` ：文本控件，包含关于一个会话的所有分派。 `CrustFrame` ：一个框架，包含一个 `Crust` 分隔窗口。

这些 `GUI` 元素可以被用在任何 `wxPython` 程序中，以提供有用的可视化内省。

`dispatcher` 模块

`dispatcher` 提供了全局信号分派服务。那意味它扮演着一个中间人，使得对象能够发送和接受消息，而无须知道彼此。所有它们需要知道的是这个正在发送的信号（通常是一个简单的字符串）。一个或多个对象可以要求这个 `dispatcher` ，当信号已发出时去通知它们，并且一个或多个对象可以告诉这个 `dispatcher` 去发送特殊的信号。

下例4.5是关于为什么 `dispatcher` 是如此有用的一个例子。因为所有的 `Py` 程序都是建造在相同的底层模块之上的，所以 `PyCrust` 和 `PyShell` 使用几乎相同的代码。这唯一的不同是， `PyCrust` 包括了一个带有额外功能的 `notebook` ，如名字空间树查看器，当命令被发送到解释器时，名字空间树查看器更新。在一个命令通过解释器时，解释器使用 `dispatcher` 发送一个信号：

例4.5 经由 `dispatcher` 模块来发送命令的代码

```
def push(self, command):
        """Send command to the interpreter to be executed.
 Because this may be called recursively, we append a new list
 onto the commandBuffer list and then append commands into
 that. If the passed in command is part of a multi-line
 command we keep appending the pieces to the last list in
 commandBuffer until we have a complete command. If not, we
 delete that last list."""

        command = str(command) # In case the command is unicode.
        if not self.more:
try:
                        del self.commandBuffer[-1]
except IndexError:
                        pass
if not self.more:
                        self.commandBuffer.append([])
                        self.commandBuffer[-1].append(command)
                        source = '\n'.join(self.commandBuffer[-1
])
                        more = self.more = self.runsource(source
)
                        dispatcher.send(signal='Interpreter.push
', sender=self,
                                command=command, more=more, sour
ce=source)
                        return more
```

`crust` 中的各有关部分和 `filling` 模块在它们的构造器中通过连接到 `dispatcher` ,自己作为信号的接受器。下例4.6显示了关于出现在 `PyCrust` 的 `Session` 标签中的 `SessionListing` 控件的源码：

例4.6 `PyCrust` `session` 标签的代码

```
class SessionListing(wx.TextCtrl):
        """Text control containing all commands for session."""

        def __init__(self, parent=None, id=-1):
                style = (wx.TE_MULTILINE | wx.TE_READONLY |wx.TE
_RICH2 | wx.TE_DONTWRAP)

                wx.TextCtrl.__init__(self, parent, id, style=sty
le)
                dispatcher.connect(receiver=self.push,signal='In
terpreter.push')

        def push(self, command, more):
                """Receiver for Interpreter.push signal."""
                if command and not more:
                        self.SetInsertionPointEnd()
                        start, end = self.GetSelection()
                if start != end:
                        self.SetSelection(0, 0)
                        self.AppendText(command + '\n')
```

注意 `SessionListing` 的接受器（ `push()` 方法）是如何忽略由解释器发送来的 `sender` 和 `source` 参数的。 `dispatcher` 非常灵活，并且只发送接受器能接受的参数。

`editor` 模块

`editor` 模块包含了出现在 `PyAlaCarte` 和 `PyAlaMode` 程序中的所有 `GUI` 编辑组件。如果你愿意在你的程序中包括一个 `Python` 源代码编辑器，那么使用在下表4.6中所说明的类。

这些类可以被使用在任何程序中以提供有用的代码风格编辑功能。

表4.6 定义在 `editor` 模块中的类

`EditerFrame` ：被 `PyAlaCarte` 用来支持一次一个文件的编辑。 `EditerFrame` 是来自 `frame` 模块的较一般的 `Frame` 类的子类。 `EditorNotebookFrame` ： `EditerFrame` 的子类，它通过增加一个 `notebook` 界面和同时编辑多个文件的能力扩展了 `EditerFrame` 。它是一个被 `PyAlaMode` 使用的 `frame` 类。 `EditorNotebook` ：这个控件被 `EditorNotebookFrame` 用来在各自的标签中显示各自的文件。 `Editor` ：管理一个 `buffer` 和与之相关的 `EditWindow` 之间的关联。 `EditWindow` ：基于 `StyledTextCtrl` 的文本编辑控件。

`filling` 模块

`filling` 模块包含所有使用户能够浏览对象的名字空间和显示关于那些对象的运行时信息的 `GUI` 控件。 定义在 `filling` 模块中的四个类的说明见下表4.7

表4.7

`FillingTree` ：基于 `wx.TreeCtrl` ， `FillingTree` 提供对象名字空间的分级树。 `FillingText` ： `editwindow.EditWindow` 的子类，用以显示当前在 `FillingTree` 所选择的对象的细节。 `Filling` ：一个 `wx.SplitterWindow` ，它的左边包括一个 `FillingTree` ，右边包括一个 `FillingText` 。 `FillingFrame` ：一个包含 `Filling` 分隔窗口的框架。双击 `filling` 树中的一项将打开一个新的 `FillingFrame` ，其中被选择的项作为树的根。

使用这些类，使你能够容量地创建 `Python` 名字空间的分级树。如果你设置你的数据为 `Python` 对象，这能够用作一个快速的数据浏览器。

`interpreter` 模块

`interpreter` 模块定义了一个 `Interpreter` 类，基于 `Python` 标准库中的 `code` 模块的 `Interactive-Interpreter` 类。除了负责发送源代码给 `Python` 外， `Interpreter` 类还负责提供自动完成列表，调用提示信息和甚至触发自动完成特性的关键码（通常是"."）。

由于这清楚的责任划分，你可以创建你自己的 `Interpreter` 的子类并传递其一个实例到 `PyCrust` `shell` ，从而代替默认的 `interpreter` 。这已经应用到了一些程序中以支持自定义评议种类，而仍然利用 `PyCrust` 环境。

`introspect` 模块

`introspect` 模块被 `Interpreter` 和 `FillingTree` 类使用。它为调用提示和命令自动完成提供了多种内省类型支持功能。下面显示了 `wx.py.introspect` 的用法，它得到一个列表对象的所有属性的名字，排除了那些以双下划线开始的属性：

```
    import wx
    L = [1, 2, 3]
    wx.py.introspect.getAttributeNames(L, includeDouble=False)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

`getAttributeNames()` 函数被 `FillingTree` 类使用以生成它的名字空间分级。理解内省模块的最好方法是关注单元测试。查看你的 `Python` 安装目录的 `Lib/site-packages/wx/py/tests` 中的 `test_introspect.py` 文件。

`shell` 模块

`shell` 模块包含出现在 `PyCrust` , `PyShell` ,和 `PyAlaMode` 定义 `Python` `shell` 界面的 `GUI` 元素。下表4.8提供了每个元素的说明。这最常用的类是 `ShellFrame` ,它是 `frame.Frame` 的子类。它包含一个 `Shell` 类的实例， `Shell` 类处理提供交互 `Python` 环境的大部分工作。

表4.8 定义在 `shell` 模块中的类

`Shell` ：`Python` `shell` 基于`wx.stc.StyleTextCtrl` 。`Shell` 子类化 `editwindow.EditWindow` ，然后使底层的文本控件的行为像一具 `Python` `shell` ，而非一个源码文件编辑器。 `ShellFacade` ：简化的与所有 `shell` 相关的功能的接口。它是半透明的，它仍然是可访问的，尽管只有一些是对 `shell` 用户可见的。 `ShellFrame` ：一个包含一个 `Shell` 窗口的框架。

`ShellFacade` 类在 `PyCrust` 的开发期间被创建，它作为在 `shell` 中访问 `shell` 对象自身时去简化事情的一个方法。当你启动 `PyCrust` 或 `PyShell` 时， `Shell` 类的实例在 `Python` `shell` 中的有效的。例如，你可以在 `shell` 提示符下调用 `shell` 的 `about()` 方法，如下所示：

```
shell.about()
Author: "Patrick K. O'Brien  pobrien@orbtech.com "
Py Version: 0.9.4
Py Shell Revision: 1.7
Py Interpreter Revision: 1.5
Python Version: 2.3.3
wxPython Version: 2.4.1.0p7
Platform: linux2
```

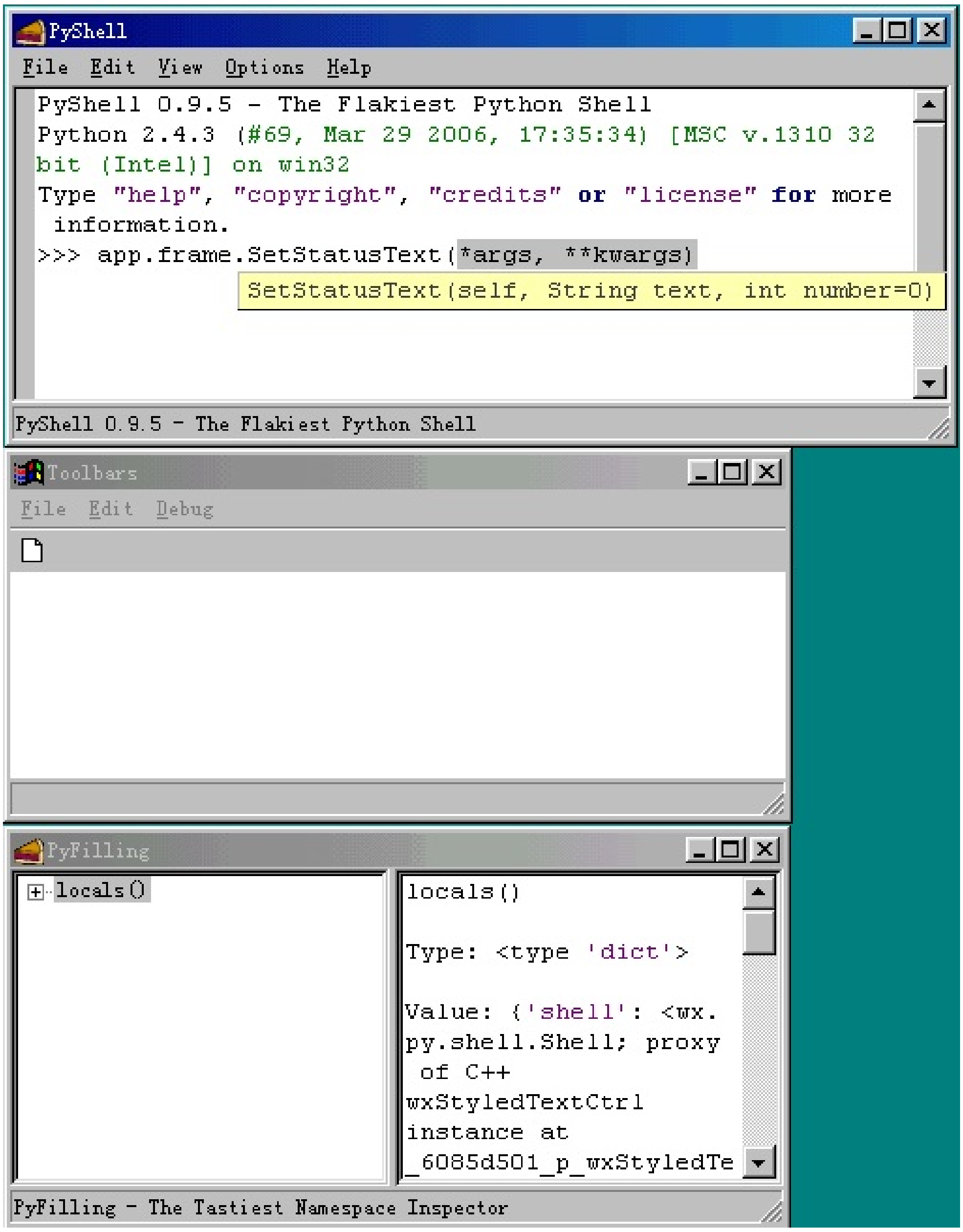
由于 `Shell` 继承自 `StyledTextCtrl` ，所以它包含超过600个属性。大部分属性对 `shell` 提示符是没用的，因此， `ShellFacade` 被创建来限制当你进入 `shell` 时出现在自动完成列表中的属性的数量。目前， `shell` 对象只显示最有用的 `shell` 属性的25个。

如何在wxPython中使用Py包中的模块?

如果你不想在你的应用程序使用一个完整的 `PyCrust` 框架，那么该怎么做呢？如果你在一个框架中仅仅只想要 `shell` 界面，而在另一个框架中要一个名字空间查看器，该怎么办呢？如果你想把它们永久添加到你的程序中以该怎么办呢？这些方案不仅是可能的，而且也是十分简单的。我们将用一个例子来说明这该怎么做来结束本章。

我们将再看一下在第2章中所创建的程序，它带有一个菜单栏，工具栏和状态栏。我们将添加一个菜单，它的一个菜单项用以显示一个 `shell` 框架，另一个用来显示一个 `filling` 框架。最后我们将把 `filling` 树的根设置给我们的主程序的框架对象。结果显示在图4.11中。

图**4.11**



下例4.7显示了修改过的源码。正如你的看到的，我们只增加了几行就实现了所要求的功能。

例4.7

```
#!/usr/bin/env python

import wx
#1 导入这些框架类
from wx.py.shell import ShellFrame
from wx.py.filling import FillingFrame
import images

class ToolbarFrame(wx.Frame):

        def __init__(self, parent, id):
                wx.Frame.__init__(self, parent, id, 'Toolbars',s
```

```
ize=(300, 200))
                panel = wx.Panel(self, -1)
                panel.SetBackgroundColour('White')
                statusBar = self.CreateStatusBar()
                toolbar = self.CreateToolBar()
                toolbar.AddSimpleTool(wx.NewId(), images.getNewB
itmap(),"New", "Long help for 'New'")
                toolbar.Realize()

                menuBar = wx.MenuBar()
                menu1 = wx.Menu()
                menuBar.Append(menu1, "  ")
                menu2 = wx.Menu()
                menu2.Append(wx.NewId(), "  ", "Copy in status b
ar")
                menu2.Append(wx.NewId(), "C ", "")
                menu2.Append(wx.NewId(), "Paste", "")
                menu2.AppendSeparator()
                menu2.Append(wx.NewId(), "  ", "Display Options"
)
                menuBar.Append(menu2, "  ")#2 创建Debug菜单及其菜单
项

                menu3 = wx.Menu()
                shell = menu3.Append(-1, " shell","Open wxPython
 shell frame")
                filling = menu3.Append(-1, " viewer","Open names
pace viewer frame")
                menuBar.Append(menu3, "  ")#3 设置菜单的事件处理器
                self.Bind(wx.EVT_MENU, self.OnShell, shell)
                self.Bind(wx.EVT_MENU, self.OnFilling, filling)

                self.SetMenuBar(menuBar)

        def OnCloseMe(self, event):
                self.Close(True)

        def OnCloseWindow(self, event):
                self.Destroy() #4 OnShell菜单项和OnFilling菜单项处
理器

        def OnShell(self, event):
                frame = ShellFrame(parent=self)
                frame.Show()

        def OnFilling(self, event):
                frame = FillingFrame(parent=self)
                frame.Show()

if __name__ == '__main__':
        app = wx.PySimpleApp()
        app.frame = ToolbarFrame(parent=None, id=-1)
        app.frame.Show()
```

```
        app.MainLoop()
```

说明：

#1 这里我们导入了 `ShellFrame` 和 `FillingFrame` 类 #2 我们添加了第三个菜单 `Debug` 到框架的菜单栏 #3 绑定一个函数给 `wx.EVT_MENU()` ，使我们能够将一个处理器与菜单项关联，以便当这个菜单项被选择时调用所关联的处理器。 #4 当用户从 `Debug` 菜单中选择 `Python shell` 时， `shell` 框架被创建，它的双亲是工具栏框架。当工具栏框架被关闭时，任何打开的 `shell` 或 `filling` 框架也被关闭。

本章小结

1、像 `wxPython` 这样的工具包本身是大而复杂的。 `GUI` 控件之间的交互并不总是直观的，整个的处理决定于事件并响应于事件，而非一个线性的执行序列。使用如 `PyCrust shell` 能够很大程度上提高你对事件驱动环境的理解。

2、 `PyCrust` 仅仅是另一个 `Python shell` ，它类似于 `IDLE` , `Boa Constructor` , `PythonWin` 和其它开发工具所包括的 `shell` 。然而， `PyCrust` 是用 `wxPython` 创建的，当你使用 `wxPython` 开发程序时，它是很有用的。尤其是，你不会有任何事件循环冲突方面的问题，并且你可以在 `PyCrust` 的 `shell` 和名字空间查看器中处理你的程序运行时的所有方面。

3、由于 `PyCrust` 是 `wxPython` 发行版的一部分，所以它随同 `wxPython` 一同被安装，包括了所有的源码。这使得 `PyCrust` 容易使用，并且减少了摸清如何在你自己的程序中提供内省功能的学习曲线。

4、另外， `Py` 包的模块化的设计，使你很容易去挑选最有益于你程序的模块，如源代码编辑、名字空间查看、或 `shell`

5、 `PyCrust` 减少了 `wxPython` 学习的曲线，并帮助你掌握你的程序运行时的细微之处。

下一章，我们将应用我们所学到的关于 `wxPython` 方面的知识，并且提供一些关于如何组织你的 `GUI` 程序的实用的建议。

第五章 绘制蓝图

众所周知， `GUI` 代码是难于阅读和维护的，并且看上去总是一塌糊涂。本章我们将讨论三个驯服你的 `UI` 代码的技术。我们将讨论重构代码以使其易于阅读、管理和维护。另一个方面是显示代码和基本的处理对象之间的处理，这也是 `UI` 程序员头痛的地方。 `MVC` （ `Model` / `View` / `Controller` ）设计模式是这样一种结构，它保持显示和数据分离以便各自的改变相互不影响。最后，我们讨论对你的 `wxPython` 代码进行单元测试的技术。尽管本章的所有例子将使用 `wxPython` ，但是其中的多数原则是可以应用到任何 `UI` 工具的，代码的设计和体系结构就是所谓的蓝图。一个深思熟虑的蓝图将使得你的应用程序建造起来更简单和更易维护。本章的建议将帮助你为你的程序设计一个可靠的蓝图。

重构如何帮我改进我的代码？

好的程序员为什么也会写出不好的界面或界面代码？这有很多原因。甚至一个简单的用户界面可能都要求很多行来显示屏幕上的所有元素。程序员通常试图用单一的方法来实现这些，这种方法迅速变得长且难于控制。此外界面代码是很容易受到不断改变的影响的，除非你对管理这些改变训练有素。由于写界面代码可能是很枯燥的，所以界面程序员经常会使用设计工具来生成代码。机器生成的代码相对于手工代码来说是很差。

原则上讲，保持 `UI` 代码在控制之下是不难的。关键是重构或不断改进现有代码的设计和结构。重构的目的是保持代码在以后易读和易于维护。下表5.1说明了在重构时需要记住的一些原则。最重要的是要记住，某人以后可能会不得不读和理解你的代码。努力让他人的生活更容易些，毕竟那有可能是你。

表**5.1** 重构的一些重要原则

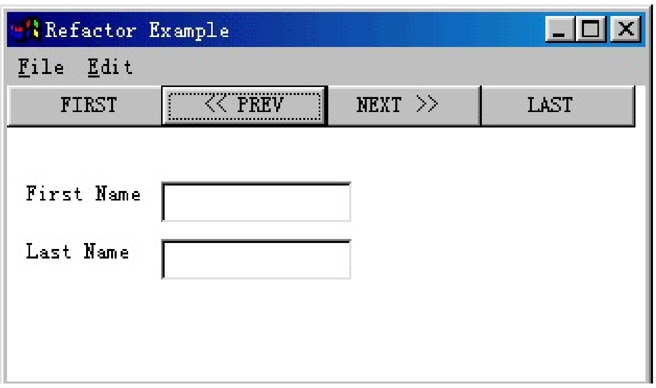
不要重复：你应该避免有多个相同功能的段。当这个功能需要改变时，这维护起来会很头痛。 一次做一件事情：一个方法应该并且只做一件事情。各自的事件应该在各自的方法中。方法应该保持短小。 嵌套的层数要少：尽量使嵌套代码不多于2或3层。对于一个单独的方法，深的嵌套也是一个好的选择。 避免字面意义上的字符串和数字：字面意义上的字符串和数字应使其出现在代码中的次数最小化。一个好的方法是，把它们从你的代码的主要部分中分离出来，并存储于一个列表或字典中。

这些原则在 `Python` 代码中特别重要。因为 `Python` 的缩进语法、小而简洁的方法是很容易去读的。然而，长的方法对于理解来说是更困难的，尤其是如果它们在一个屏幕上不能完全显示出来时。类似的， `Python` 中的深的嵌套使得跟踪代码块的开始和结尾很棘手。然而， `Python` 在避免重复方面是十分好的一种语言，特别是因为函数和方法或以作为参数传递。

一个重构的例子

为了展示给你如何在实际工作中应用这些原则，我们将看一个重构的例子。图5.1显示了一个窗口，它可用作访问微软 `Access` 类数据库的前端。

图5.1



它的布置比之前我们的所见过的那些要复杂一些。但是按现实中的应用程序的标准，它仍然十分简单。例5.1的代码的结构很差。

例**5.1** 产生图**5.1**的没有重构的代码

```
#!/usr/bin/env python

import wx

class RefactorExample(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Refactor Example',s
ize=(340, 200))
        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour("White")
        prevButton = wx.Button(panel, -1, " PREV", pos=(80, 0))
        self.Bind(wx.EVT_BUTTON, self.OnPrev, prevButton)
        nextButton = wx.Button(panel, -1, "NEXT ", pos=(160, 0))
```

```
        self.Bind(wx.EVT_BUTTON, self.OnNext, nextButton)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

        menuBar = wx.MenuBar()
        menu1 = wx.Menu()
        openMenuItem = menu1.Append(-1, "  ", "Copy in status ba
r")
        self.Bind(wx.EVT_MENU, self.OnOpen, openMenuItem)
        quitMenuItem = menu1.Append(-1, "  ", "Quit")
        self.Bind(wx.EVT_MENU, self.OnCloseWindow, quitMenuItem)
        menuBar.Append(menu1, "  ")
        menu2 = wx.Menu()
        copyItem = menu2.Append(-1, "  ", "Copy")
        self.Bind(wx.EVT_MENU, self.OnCopy, copyItem)
        cutItem = menu2.Append(-1, "C ", "Cut")
        self.Bind(wx.EVT_MENU, self.OnCut, cutItem)
        pasteItem = menu2.Append(-1, "Paste", "Paste")
        self.Bind(wx.EVT_MENU, self.OnPaste, pasteItem)
        menuBar.Append(menu2, "  ")
        self.SetMenuBar(menuBar)

        static = wx.StaticText(panel, wx.NewId(), "First Name",
                pos=(10, 50))
        static.SetBackgroundColour("White")
        text = wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1)
,
                pos=(80, 50))

        static2 = wx.StaticText(panel, wx.NewId(), "Last Name",
                pos=(10, 80))
        static2.SetBackgroundColour("White")
        text2 = wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1
),
                pos=(80, 80))

        firstButton = wx.Button(panel, -1, "FIRST")
        self.Bind(wx.EVT_BUTTON, self.OnFirst, firstButton)

        menu2.AppendSeparator()
        optItem = menu2.Append(-1, "  ", "Display Options")
        self.Bind(wx.EVT_MENU, self.OnOptions, optItem)

        lastButton = wx.Button(panel, -1, "LAST", pos=(240, 0))
        self.Bind(wx.EVT_BUTTON, self.OnLast, lastButton)

    # Just grouping the empty event handlers together
    def OnPrev(self, event): pass
    def OnNext(self, event): pass
    def OnLast(self, event): pass
    def OnFirst(self, event): pass
    def OnOpen(self, event): pass
    def OnCopy(self, event): pass
    def OnCut(self, event): pass
```

```
    def OnPaste(self, event): pass
    def OnOptions(self, event): pass

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = RefactorExample(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

根据重构原则，上面这段代码有一点是做到了，就是没有深的嵌套。其它都没有做到。 为了让你有一个关于如何调整的一个思想，我们将把所有的按钮代码分别放到各自的方法中。 下表5.2归纳了我们重构原代码应解决的问题

表5.2

原则 代码要重构的地方

不要重复几个模式不断重复，包括“增加按钮，关联一个方法”，

- “增加菜单项并关联一个方法”，“创建成对的标签/文本条目”

一次只做一件事代码做了几件事情。除了基本的框架(`frame`)设置外，它创建了菜单栏，增加了按

- 钮，增加了文本域。更糟糕的是，功能在代码中混在一起。

避免避免字面意 义上的字符串和数字在构造器中每个按钮、菜单项和文本框都有一个文字字符串和坐标常量

开始重构

例5.2中只包含了前面用于创建按键栏的代码。作为重构的第一步，我们在例5.2中把例5.1中创建按钮栏这些代码抽出来放在了它自己的方法中：

例**5.2** 按钮栏作为一个单独的方法

```
def createButtonBar(self):
        firstButton = wx.Button(panel, -1, "FIRST")
        self.Bind(wx.EVT_BUTTON, self.OnFirst, firstButton)

        prevButton = wx.Button(panel, -1, " PREV", pos=(80, 0))
        self.Bind(wx.EVT_BUTTON, , self.OnPrev, prevButton)

        nextButton = wx.Button(panel, -1, "NEXT ", pos=(160, 0))
        self.Bind(wx.EVT_BUTTON, self.OnNext, nextButton)

        lastButton = wx.Button(panel, -1, "LAST", pos=(240, 0))
        self.Bind(wx.EVT_BUTTON, self.OnLast, lastButton)
```

向上面这样把代码分离出后，所有按钮添加代码之间的共性就很容易看出来了。我们可以把添加按钮的代码写成一个公用的方法来调用 ，而避免了重复。如例5.3所示：

例**5.3** 一个公用的改进了的按钮栏方法

```
def createButtonBar(self, panel):
        self.buildOneButton(panel, "First", self.OnFirst)
        self.buildOneButton(panel, " PREV", self.OnPrev, (80, 0)
)
        self.buildOneButton(panel, "NEXT ", self.OnNext, (160, 0
))
        self.buildOneButton(panel, "Last", self.OnLast, (240, 0)
)

def buildOneButton(self, parent, label, handler, pos=(0,0)):
        button = wx.Button(parent, -1, label, pos)
        self.Bind(wx.EVT_BUTTON, handler, button)
        return button
```

例5.3代替例5.2有两个好处。第一，简短的方法和有意义的方法名使得代码的可读性更清晰了。第二，它避免了局部变量（诚然，你也可以通过使用 `ID` 来避免使用局部变量，但那容易导致重复的 `ID` 问题）。不使用局部变量是有好处的，它减少了代码的复杂程序，并且也因为这样几乎排除了通常由剪切和粘贴部分代码而忘记了改变所有变量的名字带来的错误。（在实际的应用中，你可能需要存储按钮为实例变量以备后来访问，但是本例不需要。）另外， `buildOneButton()` 方法容易放进一个工具模块中并可以在别的框架或项目中重用。

进一步重构

上面的例子，已经得到了很多的改善。但是在多处仍有许多常量。其一，就是用于定位的点坐标，当另一 个按钮被添加到按钮栏时可能使代码产生错误，尤其是新的按钮被放置在按钮栏的中间。因此让我们再往 前进一步，我们把这些字面意义上的数据从处理中分离出来。下例5.4展示了一个用于创建按钮的数据驱 动机制。

例5.4 使用分离自代码的数据创建按钮

```
def buttonData(self):
        return (("First", self.OnFirst),
                (" PREV", self.OnPrev),
                ("NEXT ", self.OnNext),
                ("Last", self.OnLast))

def createButtonBar(self, panel, yPos=0):
        xPos = 0
        for eachLabel, eachHandler in self.buttonData():
                pos = (xPos, yPos)
                button = self.buildOneButton(panel, eachLabel, e
achHandler, pos)
                xPos += button.GetSize().width

def buildOneButton(self, parent, label, handler, pos=(0,0)):
        button = wx.Button(parent, -1, label, pos)
        self.Bind(wx.EVT_BUTTON, handler, button)
        return button
```

在例5.4中，用于不同按钮的数据被存储在内嵌于 `buttonData()` 方法的元组中。所选的数据结构及常量方 法的使用不是必然的。数据也可以被存储在一个类级的变量或模块级的变量中，而非一个方法的结果，或 存储于一个外部的文件中。使用方法的好处就是，如果你的按钮数据存储在另一个地方而不是方法中的话 ，只需要改变这个方法而使它返回外部的数据。

`createButtonBar()` 方法遍历 `buttonData()` 返回的列表并创建相关数据的按钮。这个方法集依次根据列表 自动计算按钮的x坐标。这是很有帮助的，因为它保证了代码中按钮的次序与将显示在屏幕中的次序一样 ，使得代码更清晰并减少出错的机会。如果你需要将一个按钮添加到按钮栏的中间的话，你只需把数据添 加到这个列表的中间，这个代码确保了所加按钮被放置在中间。

数据的分离有其它的好处。在一个更精心制作的例子中，数据可以被存储到一个外部的资源或 `XML` 文件中 。这使得在改变界面的时候不用去关心代码，并且使国际化更容易，很容易改变文本。移除了数据以后， `createButtonBar` 方法现在成了一个公用方法了，它可以容易地在其它框架或项目中被重用。 在经过整合相同的过程，并从菜单和文本域代码中分离出数据后，所得的结果显示在如下例5.5中。

例5.5 一个重构的例子

```
#!/usr/bin/env python

import wx

class RefactorExample(wx.Frame):

        def __init__(self, parent, id):
                wx.Frame.__init__(self, parent, id, 'Refactor Ex
ample',size=(340, 200))
```

```
                panel = wx.Panel(self, -1)
                panel.SetBackgroundColour("White")
                self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
                self.createMenuBar() #简化的init方法
                self.createButtonBar(panel)
                self.createTextFields(panel)

        def menuData(self): #菜单数据
                return (("&File",
                        ("&Open", "Open in status bar", self.OnO
pen),
                        ("&Quit", "Quit", self.OnCloseWindow)),
                        ("&Edit",
                        ("&Copy", "Copy", self.OnCopy),
                        ("C&ut", "Cut", self.OnCut),
                        ("&Paste", "Paste", self.OnPaste),
                        ("", "", ""),
                        ("&Options", "DisplayOptions", self.OnOp
tions)))
#创建菜单
        def createMenuBar(self):
                menuBar = wx.MenuBar()
                for eachMenuData in self.menuData():
                        menuLabel = eachMenuData[0]
                        menuItems = eachMenuData[1:]
                        menuBar.Append(self.createMenu(menuItems
), menuLabel)
                self.SetMenuBar(menuBar)

        def createMenu(self, menuData):
                menu = wx.Menu()
                for eachLabel, eachStatus, eachHandler in menuDa
ta:
                        if not eachLabel:
                                menu.AppendSeparator()
                                continue
                        menuItem = menu.Append(-1, eachLabel, ea
chStatus)
                        self.Bind(wx.EVT_MENU, eachHandler, menu
Item)
                return menu

        def buttonData(self): #按钮栏数据
                return (("First", self.OnFirst),
                        ("<<  PREV", self.OnPrev),
                        ("NEXT >>", self.OnNext),
                        ("Last", self.OnLast))
        #创建按钮
        def createButtonBar(self, panel, yPos = 0):
                xPos = 0
                for eachLabel, eachHandler in self.buttonData():
                        pos = (xPos, yPos)
                        button = self.buildOneButton(panel, each
```

```
Label,eachHandler, pos)
                        xPos += button.GetSize().width

        def buildOneButton(self, parent, label, handler, pos=(0,
0)):
                button = wx.Button(parent, -1, label, pos)
                self.Bind(wx.EVT_BUTTON, handler, button)
                return button

        def textFieldData(self): #文本数据
                return (("First Name", (10, 50)),
                        ("Last Name", (10, 80)))
        #创建文本

        def createTextFields(self, panel):
                for eachLabel, eachPos in self.textFieldData():
                        self.createCaptionedText(panel, eachLabe
l, eachPos)

        def createCaptionedText(self, panel, label, pos):
                static = wx.StaticText(panel, wx.NewId(), label,
 pos)
                static.SetBackgroundColour("White")
                textPos = (pos[0] + 75, pos[1])
                wx.TextCtrl(panel, wx.NewId(), "", size=(100, -1
), pos=textPos)

        # 空的事件处理器放在一起
        def OnPrev(self, event): pass
        def OnNext(self, event): pass
        def OnLast(self, event): pass
        def OnFirst(self, event): pass
        def OnOpen(self, event): pass
        def OnCopy(self, event): pass
        def OnCut(self, event): pass
        def OnPaste(self, event): pass
        def OnOptions(self, event): pass
        def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
        app = wx.PySimpleApp()
        frame = RefactorExample(parent=None, id=-1)
        frame.Show()
        app.MainLoop()
```

从例5.1改变到例5.5，没有费多少力，但我们所得到的却是很多——代码非常的清楚且减少了出错的机会。代码的布置与数据的布置在逻辑上是匹配的。那些普通的做法（它们劣质的代码结构可能导致错误——如采用大量的复制和粘贴来创建新的

对象）已经被去掉。多数函数现在可以很容易地被移到一个超类或公用模块中，以保存代码便于以后继续利用。另外，数据的分离使得把这个布局作为不同数据的模板很容易，包括国际化的数据。

重构虽说完成了，但是例5.5中的代码仍然忽略了一些重要的事情：实际用户的数据。你的应用程序要做很多事依赖于处理数据响应用户要求。你的程序的结构还可以向着灵活性和稳定性方向发展。 `MVC` 模式对于管理界面和数据之间的交互是公认的标准。

如何保持模型(Model)与视图(View)分离？

最早可追溯到1970年代后期和 `Smalltalk` -80语言， `MVC` 模式大概是最早明确指出面向对象设计的模式。它是最流行的一种，被几乎所有 `GUI` 工具包所采用。 `MVC` 模式是结构化程序的标准，包括处理和显示信息。

MVC(Model-View-Controller)系统是什么？

`MVC` 系统有三个子系统。 `Model` 包含经常被调用的业务逻辑或由你的系统处理的所有数据和信息。 `View` 包含显示数据的对象， `Controller` 管理与用户的交互（ `Controller` 处于 `Model` 和 `view` 中间）。下表5.3归纳了这些组分。

表**5.3** 标准 **`MVC`** 体系的组成

组分

`Model` ：包含业务逻辑,包含所有由系统处理的数据。它包括一个针对外部存储（如一个数据库）的接口。通常模型(`model)` 只暴露一个公共的 `API` 给其它的组分。

`View` ：包含显示代码。这个窗口部件实际用于放置用户在视图中的信息。在 `wxPython` 中，处于 `wx.Window` 层级中的所有的东西都是视图(`view)` 子系统的一部分。

`Controller` ：包含交互逻辑。该代码接受用户事件并确保它们被系统处理。在 `wxPython` 中，这个子系统由 `wx.EvtHandler` 层级所代表。

在现代的 `UI` 工具包中， `View` 和 `Controller` 组分是被集成在一起的。这是因为 `Controller` 组分自身需要被显示在屏幕上，并且因为经常性的你想让显示数据的窗口部件也响应用户事件。在 `wxPython` 中，这种关系实际上已经被放置进去了（所有的 `wx.Window` 对象也都是 `wx.EvtHandler` 的子类），这意味着它们同时具有 `View` 元素和 `Controller` 元素的功能。相比之下，大部分 `web` 应用架构对于 `View` 和 `Controller` 有更严格的分离，因为其交互逻辑发生在服务器的后台。

图5.2中显示了数据和信息是如何在 `MVC` 体系中传递的。

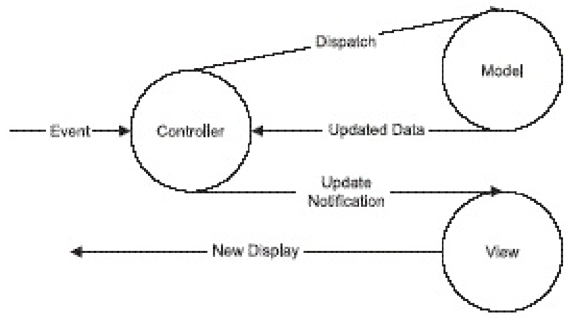


Figure 5.2 The data flow of an MVC request

一个事件通知被 `Controller` 系统处理（它把事件通知放到一个合适的地方）。如我们在第三章中所看到的，`wxPython` 使用 `wx.EvtHandler` 的方法 `ProcessEvent()` 管理这个机制。在一个严格的 `MVC` ?杓浦校奮愕拇 砥骱㉘譗赡鼙簧壁彭谝桓跑ザ赖目刂破鞠韵筠校Ï濛窃诳蚣苪嗆陨碇小? 对于事件的响应，这个模型(`model`) 对象可以对应用程序数据做一些处理。当处理完成时，模型对象发送一个更新通知。如果这儿有一个控制器(`controller`) 对象，那么该通知通常发送回这个控制器，同时这个控制器对象通知视图(`view`) 对象自我更新。在一个较小的系统或一个较简单的体系中，通知通常直接被视图对象所接受。在 `wxPython` 中，来自于模型的更新的关键在于你。你的选择包括从模型或控制器显式地引发自定义的 `wxPython` 事件，使模型维护的对象的列表接受更新通知，或使与模型关联的视图接受更新通知。

一个成功的 `MVC` 设计的关键不在于每个对象都彼此了解。相反，一个成功的 `MVC` 程序，它的不同部分之间显式地隐藏了一些东西。其目的是使系统最低限度地交互，和方法之间的明确的界定。尤其，这个 `Model` 组分应该被完全从 `View` 和 `Controller` 中脱离出来。你应该只改变那些系统而不改变你的 `Model` 类。理想上来讲，你甚至应该能够使用相同 `Model` 类来驱动非 `wxPython` 的界面，但是那很难。

从 `View` 方面，你应该能够在 `Model` 对象的实现中做改变而不改变 `View` 或 `Controller` 。而 `View` 依赖于某些公共的方法的存在，它不应该看见 `Model` 内私有的东西。无可否认，这在 `Python` 中实施是有困难的，但有一个方法可以帮助我们，那就是创建一个抽象的 `Model` 类，它定义 `View` 可以看见的 `API` 。 `Model` 的子类可以扮演一个内部的类的代理而被改变，或可以简单地自身包含内部的工作。这第一个方案更结构化些，第二个更容易实现。

下一节，我们将看一看内建于 `wxPython` 中的 `Model` 类中的一个： `wx.grid.PyGridTableBase` 。这个类使得在一个 `MVC` 设计架构中使用 `grid` 控件成为可能。这之后，我们将关注一下对于一个定制的窗口部件建造和使用定制的模型类。

一个wxPython模型：PyGridTableBase

类 `wx.grid.Grid` 是一个电子表格式样的 `wxPython` 控件。下图5.3显示了它的外观。



Figure 5.3 A sample of the wxPython grid control

这个网格(`grid`)控件有很多有趣的特点，包括能够在一个单元一个单元的基础上创建自定义的渲染器和编辑器，以及可拖拽的行和列。这些特性将在第十三章中做更详细的讨论。在这一章，我们将针对基础和展示如何使用一个模型去填充网格。例5.6显示在一个网格中设置单元值的简单的非模型方法。在此例中，网格中的值是1984年芝加哥小熊队的阵容。

例**5.6** 填充网格（没有使用模型）

```
import wx
import wx.grid

class SimpleGrid(wx.grid.Grid):
        def __init__(self, parent):
                wx.grid.Grid.__init__(self, parent, -1)
                self.CreateGrid(9, 2)
                self.SetColLabelValue(0, "First")
                self.SetColLabelValue(1, "Last")
                self.SetRowLabelValue(0, "CF")
                self.SetCellValue(0, 0, "Bob")
                self.SetCellValue(0, 1, "Dernier")
                self.SetRowLabelValue(1, "2B")
                self.SetCellValue(1, 0, "Ryne")
                self.SetCellValue(1, 1, "Sandberg")
                self.SetRowLabelValue(2, "LF")
                self.SetCellValue(2, 0, "Gary")
                self.SetCellValue(2, 1, "Matthews")
                self.SetRowLabelValue(3, "1B")
                self.SetCellValue(3, 0, "Leon")
                self.SetCellValue(3, 1, "Durham")
                self.SetRowLabelValue(4, "RF")
                self.SetCellValue(4, 0, "Keith")
                self.SetCellValue(4, 1, "Moreland")
                self.SetRowLabelValue(5, "3B")
                self.SetCellValue(5, 0, "Ron")
                self.SetCellValue(5, 1, "Cey")
                self.SetRowLabelValue(6, "C")
                self.SetCellValue(6, 0, "Jody")
                self.SetCellValue(6, 1, "Davis")
                self.SetRowLabelValue(7, "SS")
                self.SetCellValue(7, 0, "Larry")
                self.SetCellValue(7, 1, "Bowa")
                self.SetRowLabelValue(8, "P")
                self.SetCellValue(8, 0, "Rick")
                self.SetCellValue(8, 1, "Sutcliffe")

class TestFrame(wx.Frame):
        def __init__(self, parent):
                wx.Frame.__init__(self, parent, -1, "A Grid",siz
e=(275, 275))
                grid = SimpleGrid(self)

if __name__ == '__main__':
        app = wx.PySimpleApp()
        frame = TestFrame(None)
        frame.Show(True)
        app.MainLoop()
```

在例5.6中，我们产生了`SimpleGrid`类，它是`wxPython`类`wx.grid.Grid`的子类。如前所述，`wx.grid.Grid`有很多种方法，这我们以后再讨论。现在，我们只关心方法`SetRowLabelValue()`，`SetColLabelValue()`和`SetCellValue()`，它们实际上设置显示在网格中的值。通过对比图5.3和例5.6你可以明白，`SetCellValue()`方法要求一个行索引、一个列索引和一个值。而其它两个方法要求一个索引和一个值。

上面的代码使用了`set`*的方法直接把值赋给了网格。然而如果对于一个较大的网格使用这种方法，代码将冗长乏味，并很容易导致错误的出现。即使我们创建公用程序以减轻负担，但是根据重构的原则，代码仍有问题。数据与显示混在一起，对于将来代码的修改是困难的，如增加一列或更换数据。

解决的答案就是`wx.grid.PyGridTableBase`。根据之前我们所见过的其它的类，前缀`Py`表明这是一个封装了C++类的特定的`Python`类。就像我们在第三章中所见的`PyEvent`类，`PyGridTableBase`的实现是基于简单封装一个`wxWidgets` C++类，这样的目的是使得能够继续声明该类(`Python`形式的类)的子类。`PyGridTableBase`对于网格是一个模型类。也就是说，网格对象可能使用`PyGridTableBase`所包含的方法来绘制自身，而不必了解有关绘制数据的内部结构。

`PyGridTableBase`的方法

`wx.grid.PyGridTableBase`有一些方法，它们中的许多你不会用到。这个类是抽象的，并且不能被直接实 例化。每次你创建一个`PyGridTableBase`时，有五个必要的方法必须被定义。表5.4说明了这些方法。

表5.4 `wx.grid.PyGridTableBase`的必须的方法

`GetNumberRows()`：返回一个表明`grid`中行数的整数。

`GetNumberCols()`：返回一个表明`grid`中列数的整数。

`IsEmptyCell(row`，`col)`：如果索引(`row`，`col)`所表示的单元是空的话，返回`True`。

`GetValue(row`，`col)`：返回显示在单元(`row`，`col)`中的值。

`SetValue(row`，`col`，`value)`：设置单元(`row`，`col)`中的值。如果你想要只读模式，你仍必须包含这个方法，但是你可以在该函数中使用`pass`。

表(`table)`通过使用网格(`grid)`的`SetTable()`方法被附加在`grid`上。在属性被设置后，`grid`对象将调用表 的方法来得到它绘制网格所需要的信息。`grid`不再显式使用`grid`的方法来设置值。

使用`PyGridTableBase`

一般情况下，有两种使用`PyGridTableBase`的方法。你可以显式地使你的模型类是`PyGridTableBase`的子类，或你可以创建一个单独的`PyGridTableBase`的子类，它关联你的实际的模型类。当你的数据不是太复杂 的时候，第一种方案较简单并且直观。第二种方案需要对模型和视图做很好的分离，如果你的数据复杂的话，

这第二种方案是更好的。如果你有一个预先存在的数据类，你想把它用于 `wxPython` ，那么这第二种方案也是更好的，因为这样你可以创建一个表而不用去改变已有的代码。在下面一节我们将展示包含这两种方案的一个例子。

使用 **`PyGridTableBase`** ：特定于应用程序（不通用）的子类

我们的第一个例子将使用 `PyGridTableBase` 的一个特定于应用程序的子类作为我们的模型。由于我们小熊 队阵容的相对简单些，所以我们使用它。我们把这些数据组织到一个派生自 `PyGridTableBase` 的类。我们 把这些实际的数据配置在一个二维 `Python` 列表中，并且配置另外的方法来从列表中读。下例5.7展示了生 成自一个模型类的小熊队的阵容。

例**5.7** 生成自 **`PyGridTableBase`** 模型的一个表

```
import wx
import wx.grid

class LineupTable(wx.grid.PyGridTableBase):

        data = (("CF", "Bob", "Dernier"), ("2B", "Ryne", "Sandberg"),
                ("LF", "Gary", "Matthews"), ("1B", "Leon", "Durham"),
                ("RF", "Keith", "Moreland"), ("3B", "Ron", "Cey"),
                ("C", "Jody", "Davis"), ("SS", "Larry", "Bowa"),
                ("P", "Rick", "Sutcliffe"))

        colLabels = ("Last", "First")

        def __init__(self):
                wx.grid.PyGridTableBase.__init__(self)

        def GetNumberRows(self):
                return len(self.data)

        def GetNumberCols(self):
                return len(self.data[0]) - 1

        def GetColLabelValue(self, col):
                return self.colLabels[col]

        def GetRowLabelValue(self, row):
                return self.data[row][0]

        def IsEmptyCell(self, row, col):
                return False

        def GetValue(self, row, col):
                return self.data[row][col + 1]

        def SetValue(self, row, col, value):
```

```
                pass

class SimpleGrid(wx.grid.Grid):
        def __init__(self, parent):
                wx.grid.Grid.__init__(self, parent, -1)
                self.SetTable(LineupTable()) #设置表

class TestFrame(wx.Frame):
        def __init__(self, parent):
                wx.Frame.__init__(self, parent, -1, "A Grid",siz
e=(275, 275))
                grid = SimpleGrid(self)

if __name__ == '__main__':
        app = wx.PySimpleApp()
        frame = TestFrame(None)
        frame.Show(True)
        app.MainLoop()
```

在例5.7中，我们已经定义了所有必须的`PyGridTableBase`方法，并加上了额外的方法`GetColLabelValue()`和`GetRowLabelValue()`。希望你不要对这两个额外的方法感到诧异，这两个额外的方法使得表(`table`)能够分别指定行和列的标签。在重构一节中，使用模型类的作用是将数据与显示分开。在本例中，我们已经把数据移入了一个更加结构化的格式，它能够容易地被分离到一个外部文件或资源中（数据库容易被增加到这里）。

使用**`PyGridTableBase`**：一个通用的例子

实际上，上面的例子很接近一个通用的能够读任何二维`Python`列表的表了。下列5.8展示这通用的模型的 外观：

例**5.8** 一个关于二维列表的通用的表

```
import wx
import wx.grid

class GenericTable(wx.grid.PyGridTableBase):

    def __init__(self, data, rowLabels=None, colLabels=None):
        wx.grid.PyGridTableBase.__init__(self)
        self.data = data
        self.rowLabels = rowLabels
        self.colLabels = colLabels

    def GetNumberRows(self):
        return len(self.data)

    def GetNumberCols(self):
        return len(self.data[0])

    def GetColLabelValue(self, col):
        if self.colLabels:
            return self.colLabels[col]

    def GetRowLabelValue(self, row):
        if self.rowLabels:
            return self.rowLabels[row]

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        return self.data[row][col]

    def SetValue(self, row, col, value):
        pass
```

`GenericTable` 类要求一个数据的二维列表和一个可选的行和列标签列表。这个类适合被导入任何 `wxPython` 程序中。使用一个做了微小改变的格式，我们现在可以使用这通用的表来显示阵容，如下例5.9所示：

例**5.9** 使用这通用的表来显示阵容

```
import wx
import wx.grid
import generictable

data = (("Bob", "Dernier"), ("Ryne", "Sandberg"),
        ("Gary", "Matthews"), ("Leon", "Durham"),
        ("Keith", "Moreland"), ("Ron", "Cey"),
        ("Jody", "Davis"), ("Larry", "Bowa"),
        ("Rick", "Sutcliffe"))

colLabels = ("Last", "First")
rowLabels = ("CF", "2B", "LF", "1B", "RF", "3B", "C", "SS", "P")

class SimpleGrid(wx.grid.Grid):
    def __init__(self, parent):
        wx.grid.Grid.__init__(self, parent, -1)
        tableBase = generictable.GenericTable(data, rowLabels,
                colLabels)
        self.SetTable(tableBase)

class TestFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "A Grid",
                size=(275, 275))
        grid = SimpleGrid(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TestFrame(None)
    frame.Show(True)
    app.MainLoop()
```

使用 `PyGridTableBase` ：一个独立的模型类

至于避免重复性，有另一种使用 `PyGridTableBase` 的方法值得在这展示给大家。这就是我们早先提到的第 二种方案，数据在一个单独的模型类中，通过 `PyGridTableBase` 来访问。 `Python` 的自我检查功能在这是非 常有用的，使你能够在每列显示一个属性的列表，然后使用内建函数 `getattr()` 去获取实际的值。在这种 情况下，模型要求一个元素的列表。在 `wxPython` 中，使用单独的模型对象结构化你的程序有一个大的优势 。在通常的情形下，对于一个 `grid` ，你只能调用 `SetTable()` 一次，如果你想去改变表，你需要创建一个新 的 `grid` ，那是烦人的。然而，在接下来的例子中，你的 `PyGridTableBase` 仅存储了对于你的实际数据类的 实例的引用，这样一来，以后你就只需通过改变表中基本的数据对象，就可以更新表中的数据为新的数据 了。

下例5.10展示使用了关于阵容条目的单独的数据类的 `PyGridTableBase` 。我们省去了框架的另一列表和数据 创建，它们是与前一例子十分类似的。

例**5.10** 使用了一个自定义的数据类的阵容显示表

```
import wx
import wx.grid

class LineupEntry:

    def __init__(self, pos, first, last):
        self.pos = pos
        self.first = first
        self.last = last

class LineupTable(wx.grid.PyGridTableBase):

    colLabels = ("First", "Last") # 列标签
    colAttrs = ("first", "last") #1 属性名

    def __init__(self, entries): #2 初始化模型
        wx.grid.PyGridTableBase.__init__(self)
        self.entries = entries

    def GetNumberRows(self):
        return len(self.entries)

    def GetNumberCols(self):
        return 2

    def GetColLabelValue(self, col):
        return self.colLabels[col] #读列标签

    def GetRowLabelValue(self, col):
        return self.entries[row].pos #3 读行标签

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        entry = self.entries[row]
        return getattr(entry, self.colAttrs[col]) #4 读属性值

    def SetValue(self, row, col, value):
        pass
```

说明：

#1：这个列表包含了一些属性，它们被引用去按列地显示每列的值。

#2：这个模型要求一个条目的列表，每个条目都是`LineupEntry`的一个实例。（这里我们没有做任何的错误检查）。

#3：要得到行头的标签，我们查看条目的`pos`属性。

#4：第一步是根据行来得到正确的条目。所要求的属性来自于#1中的列表，然后 `getattr()` 被用来引用实际的值。这个机制是可扩展的，即使是在你不知道该名字是否引用一个属性或方法的情况下，你也可以通 过检查 `object . attribute` 来看是否其可调用。如果可调用，那么使用通常的 `Python` 函数语法来调用它 ，并返回它的值。

`grid` 类是 `wxPython` 已有的一个有价值的模型组件来帮助你结构化你的应用程序的一个例子。下一节我们将 讨论如何为别的 `wxPython` 对象创建模型组件。

自定义模型

创建你的模型对象所基于的基本思想是简单的。首先构造你的数据类而不要担心它们将如何被显示。然后 为数据类作一个公共接口，该接口对显示对象是能够被访问的。很明显，这个工程的大小和复杂性将决定 这个公共声明的形式如何。在一个小的工程中，使用简单的对象，可能足够做简单的事件和使视图对象能 够访问该模型的属性。在一个更复杂的对象中，对于这种使用你可能想定义特殊的方法，或创建一个分离 的模型类，该类是视图唯一看到的东西（正如我们在例5.10所做的）。

为了使视图由于模型中的改变而被得到通知，你也需要某种机制。例5.11展示了一个简单的——一个抽象 的基类，你可以用它作为你的模型类的双亲。你可以把这看成 `PyGridTableBase` 用于当显示不是一个网格(`grid)` 时的一个类似情况。

例**5.11** 用于更新视图的一个自定义的模型

```
class AbstractModel(object):

        def __init__(self):
                self.listeners = []

        def addListener(self, listenerFunc):
                self.listeners.append(listenerFunc)

        def removeListener(self, listenerFunc):
                self.listeners.remove(listenerFunc)

        def update(self):
                for eachFunc in self.listeners:
                        eachFunc(self)
```

我们这里的 `listener` 应该是可调用的对象，它需要 `self` 作为参数（ `eachFunc(self)` ）——很明显， `self` 的 实际的类可以是不同的，因此你的 `listenter` 很灵活了。同样，我们已经将 `AbstractModel` 配置成一个 `Python` 的新型的类，事实上它是 `object` 的子类。因此本例要求 `Python` 的版本是 `Python2.2` 或更高。

我们如何使用这个抽象的类呢？图5.4显示了一个新的窗口，类似于我们本章早先讲重构时所显示的窗口 。这个窗口简单。其中文本框是只读的。敲击上面的任一按钮将在文本框中显示相应的字符。

显示这个窗口的程序使用了一个简单的 `MVC` 结构。按钮的处理器方法引起这个模型中的变化，模型中的更 新导致文本域的改变。例5.12展示了这个细节。

例**5.12**

```
#!/usr/bin/env python

import wx
import abstractmodel

class SimpleName(abstractmodel.AbstractModel):

    def __init__(self, first="", last=""):
        abstractmodel.AbstractModel.__init__(self)
        self.set(first, last)

    def set(self, first, last):
        self.first = first
        self.last = last
        self.update()   #1 更新

class ModelExample(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Flintstones',
                size=(340, 200))
        panel = wx.Panel(self)
        panel.SetBackgroundColour("White")
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
        self.textFields = {}
        self.createTextFields(panel)
#-------------------------------
    #2 创建模型
        self.model = SimpleName()
        self.model.addListener(self.OnUpdate)
        #-------------------------------
        self.createButtonBar(panel)

    def buttonData(self):
        return (("Fredify", self.OnFred),
                ("Wilmafy", self.OnWilma),
                ("Barnify", self.OnBarney),
                ("Bettify", self.OnBetty))

    def createButtonBar(self, panel, yPos = 0):
        xPos = 0
        for eachLabel, eachHandler in self.buttonData():
            pos = (xPos, yPos)
            button = self.buildOneButton(panel, eachLabel, eachH
andler, pos)
```

```
            xPos += button.GetSize().width

    def buildOneButton(self, parent, label, handler, pos=(0,0)):
        button = wx.Button(parent, -1, label, pos)
        self.Bind(wx.EVT_BUTTON, handler, button)
        return button

    def textFieldData(self):
        return (("First Name", (10, 50)),
                ("Last Name", (10, 80)))

    def createTextFields(self, panel):
        for eachLabel, eachPos in self.textFieldData():
            self.createCaptionedText(panel, eachLabel, eachPos)

    def createCaptionedText(self, panel, label, pos):
        static = wx.StaticText(panel, wx.NewId(), label, pos)
        static.SetBackgroundColour("White")
        textPos = (pos[0] + 75, pos[1])
        self.textFields[label] = wx.TextCtrl(panel, wx.NewId(),
                "", size=(100, -1), pos=textPos,
                style=wx.TE_READONLY)

    def OnUpdate(self, model): #3 设置文本域
        self.textFields["First Name"].SetValue(model.first)
        self.textFields["Last Name"].SetValue(model.last)
    #-------------------------------------------
    #4 响应按钮敲击的处理器
    def OnFred(self, event):
        self.model.set("Fred", "Flintstone")

    def OnBarney(self, event):
        self.model.set("Barney", "Rubble")

    def OnWilma(self, event):
        self.model.set("Wilma", "Flintstone")

    def OnBetty(self, event):
        self.model.set("Betty", "Rubble")
    #---------------------------------------------
    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ModelExample(parent=None, id=-1)
    frame.Show()
    app.MainLoop()
```

说明：

#1：这行执行更新

#2：这两行创建这个模型对象，并且把`OnUpdate()`方法注册为一个`listener`。现在当更新被调用时，`OnUpdate()`方法将被调用。

#3：`OnUpdate()`方法本身简单地使用模型更新后的值来设置文本域中的值。该方法的代码中可以使用`self.model`这个实例来代替`model`（它们是同一个对象）。使用方法作为参数，代码是更健壮的，在这种情况下，同样的代码可以监听多个对象。

#4：按钮敲击的处理器改变模型对象的值，它触发更新。

在这样一个小的例子中，使用模型更新机制似乎有点大才小用了。为什么按钮处理器不能直接设置文本域的值呢。然而，当这个模型类存在一个更复杂的内部状况和处理时，这个模型机制就变得更有价值了。例如，你将能够将内部的分配从一个`Python`字典改变为一个外部的数据库，而不在视图中做任何改变。 假如你正在处理一个已有的类，而不能或不愿对其做改变，那么`AbstractModel`可以用作该类的代理，方法与例5.10中的阵容所用的方法大致相同。

另外，`wxPython`包含两个单独的类似`MVC`更新机制的实现，它们比我们这里说明的这个有更多的特性。第一个是模块`wx.lib.pubsub`，它在结构上与我们先前给出的类`AbstractModel`十分相似。名为`Publisher`的模型类使得对象能够监听仅特定类型的消息。另一个更新系统是`wx.lib.evtmgr.eventManager`，它建立在`pubsub`之上，并且有一些额外的特性，包括一个更精心的面向对象的设计和事件关联的连接或去除的易用性。

如何对一个GUI程序进行单元测试？

好的重构和`MVC`设计模式的一个主要的好处是，它使得使用“单元测试”来验证你程序的性能更容易了。单元测试是对你的程序的单个的特定功能的测试。由于重构和`MVC`设计模式两者的使用，使得你的程序被分成了小的块，因此你更容易针对你程序的个别的部分写特定的单元测试。当重构的时候，结合使用单元测试是特别有用的，因为完整的单元测试使得在你移动你的代码后，你能够检验你是否引入了任何错误。

接下来的难题是在单元测试中如何测试`UI`代码。测试一个模型是相对简单的，因为模型的大部分功能不依赖于用户的输入。测试界面本身的功能比较困难，因为界面的行为依赖于用户的行为，而用户的行为又是难以封装在内的。在这一节，我们将给你展示如何在`wxPython`中使用单元测试。尤其是在单元测试期间手工产生事件去触发行为的用法。

unittest模块

当写用户测试的时候，使用已有的测试引擎来节省减少重复的写代码的运行你的测试是有帮助的。自2.1版以来，`Python`已发布了`unittest`模块。`unittest`模块使用一名为`PyUnit`的测试框架（参见`http:`//`pyunit.sourceforge.net`/）。`PyUnit`模块由`Test`,`TestCase`,`TestSuite`组成。下表5.5说明了这三个组成。

表5.5

`Test` ：被 `PyUnit` 引擎调用的一个单独的方法。根据约定，一个测试方法的名字以 `test` 开头。测试方法通常执行一些代码，然后执行一个或多个断定语句来测试结果是否是预期的。

`TestCase` ：一个类，它定义了一个或多个单独的测试，这些测试共享一个公共的配置。这个类定义在 `PyUnit` 中以管理一组这样的测试。 `TestCase` 在测试前后对每个测试都提供了公共配置支持，确保每个测试分别运行。 `TestCase` 也定义了一些专门的断定方法，如 `assertEqual` 。

`TestSuite` ：为了同时被执行而组合在一起的一个或多个 `test` 方法或 `TestCase` 对象。当你告诉 `PyUnit` 去执行测试时，你传递给它一个 `TestSuite` 对象去执行。

单个的 `PyUnit` 测试可能有三种结果： `success` （成功）, `failure` （失败）, 或 `error` （错误）。 `success` 表明测试完成，所有的断定都为真（通过），并且没有引发错误。也就是说得到了我们所希望的结果。 `Failure` 和 `error` 表明代码存在问题。 `failure` 意味着你的断定之一返回 `false` ，表明代码执行成功了，但是没有做你预期的事。 `error` 意味着测试执行到某处，触发了一个 `Python` 异常，表明你的代码没有运行成功。在单个的测试中， `failure` 或 `error` 一出现，整个测试就终止了，即使在代码中还有多个断定要测试，然后测试的执行将移到到下一个单个的测试。

一个unittest范例

下例5.13展示了一个使用 `unittest` 模块的范例，其中对例5.12中的模型例子进行测试。

例5.13 对模型例子进行单元测试的一个范例

```
import unittest
import modelExample
import wx

class TestExample(unittest.TestCase): #1 声明一个TestCase

    def setUp(self): #2 为每个测试所做的配置
        self.app = wx.PySimpleApp()
        self.frame = modelExample.ModelExample(parent=None, id=-
1)

    def tearDown(self): #3 测试之后的清除工作
        self.frame.Destroy()

    def testModel(self): #4 声明一个测试(Test)
        self.frame.OnBarney(None)
        self.assertEqual("Barney", self.frame.model.first,
                msg="First is wrong") #5 对于可能的失败的断定
        self.assertEqual("Rubble", self.frame.model.last)

def suite(): #6 创建一个TestSuite
    suite = unittest.makeSuite(TestExample, 'test')
    return suite

if __name__ == '__main__':
    unittest.main(defaultTest='suite') #7 开始测试
```

说明：

#1：声明 `unittest.TestCase` 的一个子类。为了最好的使每个测试相互独立，测试执行器为每个测试创建该类的一个实例。

#2： `setUp()` 方法在每个测试被执行前被调用。这使得你能够保证每个对你的应用程序的测试都处在相同的状态下。这里我们创建了一个用于测试的框架(`frame)` 的实例。

#3 ： `tearDown()` 方法在每个测试执行完后被调用。这使得你能够做一些清理工作，以确保从一个测试转到另一个测试时系统状态保持一致。通常这里包括重置全局数据，关闭数据库连接等诸如此类的东东。这里我们对框架调用了 `Destroy()` ，以强制性地使 `wxWidgets` 退出，并且为下一个测试保持系统处在一个良好的状态。

#4 ：测试方法通常以 `test` 作为前缀，尽管这处于你的控制之下（看#6）。测试方法不要参数。我们这里的测试方法中，通过调用 `OnBarney` 事件处理器方法来开始测试行为。

#5 ：这行使用 `assertEqual()` 方法来测试模型对象的改变是否正确。 `assertEqual()` 要两个参数，如果这两个参数不相等，则测试失败。所有的 `PyUnit` 断定方法都有一个可选的参数 `msg` ，如果断定失败则显示 `msg(msg` 的默认值几乎够表达意思了）

#6： 这个方法通过简单有效的机制创建一组测试。 `makeSuite()` 方法要求一个 `Python` 的类的对象和一个字符串前缀作为参数，并返回一组测试（包含该类中所有前缀为参数“前缀”的方法）。还有其它的机制，它们使得可以更明确设置测试组中的内容，但是 `makeSuite()` 方法通过足够了。我们这里写的 `suite()` 方法是一个样板模板，它可被用在你的所有测试模块中。

#7 ：这行调用了 `PyUnit` 的基于文本的执行器。参数是一个方法的名字（该方法返回一测试组）。然后 `suite` 被执行，并且结果被输出到控制台。如果你想使用 `GUI` 测试执行器，那么这行调用应使用 `unittest.TextTestRunner` 的方法而非 `unittest.main` 。

在控制台中 `PyUnit` 测试的结果如下：

.

`Ran` 1 `test` `in` 0.190s

`OK`

这是一个成功的测试。第一行的"."号表明测试成功。每个测试都得到一个字符并显示在这行。"."表明成功，"F"表明失败，"E"表明错误。然后是一个简单的列表，其中包含测试的数量、总的测试时间和 `OK` ， `OK` 表明所有测试通过。

对于一个失败或错误的测试，你将得到一堆跟踪提示（显示了 `Python` 得到的错误处的情况）。比如，如果你将#5改为 `self.assertEqual("Fife", self.frame.model.first)` ，我们将得到如下结果：

```
F
============================================================
======
FAIL: testModel (__main__.TestExample)
------------------------------------------------------------
------
Traceback (most recent call last):
File "C:\wxPyBook\book\1\Blueprint\testExample.py", line 18, in
testModel
self.assertEqual("Fife", self.frame.model.last)
File "c:\python23\lib\unittest.py", line 302, in failUnlessEqual
raise self.failureException, \
AssertionError: 'Fife' != 'Rubble'

------------------------------------------------------------
------
Ran 1 test in 0.070s

FAILED (failures=1)
```

"F"表明了失败，“ `testModel` ”是产生失败的方法名，下面的跟踪显示出18号上的断定失败，以及失败的原因。你一般需要根据这些去找到产生失败的实际位置。

测试用户事件

当然，上面的测试还不完整。我们还可以对框架中的 `TextField` 在模型更新后，其中值的更新情况进行测试。这个测试是很简单的。另一个你可能想要做的测试是，自动生成按钮敲击事件，以及确保正确的处理器被调用。这个测试有点难度。下例5.14展示了一个例子：

例5.14 生成一个用户事件的测试

```
def testEvent(self):
panel = self.frame.GetChildren()[0]
        for each in panel.GetChildren():
                if each.GetLabel() == "Wilmafy":
                        wilma = each
                        break
        event = wx.CommandEvent(wx.wxEVT_COMMAND_BUTTON_CLICKED,
  wilma.GetId())
        wilma.GetEventHandler().ProcessEvent(event)
        self.assertEqual("Wilma", self.frame.model.first)
        self.assertEqual("Flintstone", self.frame.model.last)
```

本例开始的几行寻找一个适当的按钮（这里是" `Wilmafy` "按钮）。由于我们没有显式地把这些按钮存储到变量中，所以我们就需要遍历 `panel` 的孩子列表，直到我们找到正确的按钮。接下来的两行创建用以被按钮发送的 `wx.CommandEvent` 事件，并发送出去。参数 `wx.wxEVT_COMMAND_BUTTON_CLICKED` 是一个常量，它表示一个事件类型，是个整数值，它被绑定到 `EVT_BUTTON` 事件绑定器对象。（你能够在 `wx.py` 文件中发现这个整数常量）。 `wilma.GetId()` 的作用是设置产生该事件的按钮 `ID` 。至此，该事件已具有了实际 `wxPython` 事件的所有相关特性。然后我们调用 `ProcessEvent()` 来将该事件发送到系统中。如果代码按照计划工作的话，那么模型的 `first` 和 `last` 中的名字将被改变为“ `Wilma` ”和“ `Flintstone` ”。

通过生成事件，你能够从头到尾地测试你的系统的响应性。理论上，你可以生成一个鼠标按下和释放事件以确保响应按钮敲击的按钮敲击事件被创建。但是实际上，这不会工作，因为低级的 `wx.Events` 没有被转化为本地系统事件并发送到本地窗口部件。然而，当测试自定义的窗口部件时，可以用到类似于第三章中两个按钮控件的处理。此类单元测试，对于你的应用程序的响应性可以给你带来信心。

本章小结

1、众所周知， `GUI` 代码看起来很乱且难于维护。这一点可以通过一点努力来解决，当代码以后要变动时，我们所付出的努力是值得的。

2、重构是对现存代码的改进。重构的目的有：避免重复、去掉无法理解的字面值、创建短的方法（只做一件事情）。为了这些目标不断努力将使你的代码更容易去读和理解。另外，好的重构也几乎避免了某类错误的发生（如剪切和粘贴导致的错误）。

3、把你的数据从代码中分离出来，使得数据和代码更易协同工作。管理这种分离的标准机制是 `MVC` 机制。用 `wxPython` 的术语来说， `V(View)` 是 `wx.Window` 对象，它显示你的数据； `C(Controller)` 是 `wx.EvtHandler` 对象，它分派事件； `M(Model)` 是你自己的代码，它包含被显示的信息。

4、或许 `MVC` 结构的最清晰的例子是 `wxPython` 的核心类中的 `wx.grid.PyGridTableBase` ，它被用于表示数据以在一个 `wx.grid.Grid` 控件中显示。表中的数据可以来自于该类本身，或该类可以引用另一个包含相关数据的对象。

5、你可以使用一个简单的机制来创建你自己的 `MVC` 设置，以便在模型被更新时通知视图(`view)` 。在 `wxPython` 中也有现成的模块可以帮助你做这样的事情。

6、单元测试是检查你的程序的正确性的一个好的方法。在 `Python` 中， `unittest` 模块是执行单元测试的标准方法中的一种。使一些包，对一个 `GUI` 进行单元测试有点困难，但是 `wxPython` 的可程序化的创建事件使得这相对容易些了。这使得你能够从头到尾地去测试你的应用程序的事件处理行为。

在下一章中，我们将给你展示如何去建造一个小型的应用程序以及如何去做一些事情，这些对你将建造的 `wxPython` 应用程序将是通用的。

第六章 使用wxPython基本构件

即使是一个简单的 `wxPython` 程序也需要使用标准的元素，诸如菜单和对话框。这儿有对于任一 `GUI` 应用程序的基本的建造部件。使用这些建造部件，还有像启动画面、状态栏或关于框等这些窗口部件，它们给你提供了一个更友好的用户环境，并且给了你的应用程序一个专业的感观。为了要结束本书的第一部分，我们将指导你通过一个程序的创建来使用所有所学的部分。我们将建造一个简单的绘画程序，然后添加这些建造部件元素并说明使用它们时的一些问题。我们将巩固前面章节的基本原理和概念，并且最后我们将产生这个简单但专业的应用程序。本章位于先前基本概念章节和后面对 `wxPython` 功能更详细讨论的2、3部分之间。

我们在本章将建造的这个应用程序基本上基于 `wxPython` / `samples` 目录中的涂鸦例子。这个应用程序是一个非常简单的绘画程序，当鼠标左键按下时它跟踪鼠标指针，并画线。图6.1显示了一个简单的初始绘画窗口。

图6.1

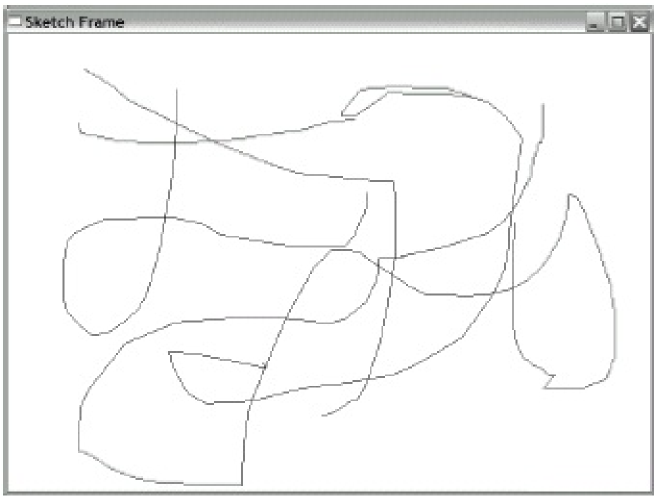


Figure 6.1 A simple sketch window, with no further decorations

我们之所以要选择这样一个绘画例子，是因为它是十分简单的程序，它演示了在创建更复杂的应用程序时所引出的许多问题。在本章，我们将给你展示如何在屏幕上画线、添加状态栏、工具样以及菜单栏。你将会看到如何使用通用对话框，如文件选择器和颜色选择器。我们将使用 `sizer` 来布置窗口部件，并且我们也将增加一个关于框和一个启动画面。本章的最后，你将有一个很好看的绘画程序。

在屏幕上绘画

你的绘画程序的首先的工作是勾画线条并显示出来。像其它的 `GUI` 工具一样， `wxPython` 提供了一套独立于设备的工具用于绘画。下面，我们将讨论如何在屏幕上绘画。

如何在屏幕上绘画

要在屏幕上绘画，我们要用到一个名为 `device` `context` （设备上下文）的 `wxPython` 对象。设备上下文代表抽象的设备，它对于所有的设备有一套公用的绘画方法，所以对于不同的设备，你的代码是相同的，而不用考虑你所在的具体设备。设备上下文使用抽象的 `wxPython` 的类 `wx.DC` 和其子类来代表。由于 `wx.DC` 是抽象的，所以对于你的应用程序，你需要使用它的子类。

使用设备上下文

表6.1显示了 `wx.DC` 的子类及其用法。设备上下文用来在 `wxPython` 窗口部件上绘画，它应该是局部的，临时性的，不应该以实例变量、全局变量或其它形式在方法调用之间保留。在某些平台上，设备上下文是有限的资源，长期持有 `wx.DC` 可能导致你的程序不稳定。由于 `wxPython` 内部使用设备上下文的方式，对于在窗口部件中绘画，就存在几个有着细微差别的 `wx.DC` 的子类。第十二章将更详细地说明这些差别。

表6.1

`wx.BufferedDC` ：用于缓存一套绘画命令，直到命令完整并准备在屏幕上绘画。这防止了显示中不必要的闪烁。

`wx.BufferedPaintDC` ：和 `wx.BufferedDC` 一样，但是只能用在一个 `wx.PaintEvent` 的处理中。仅临时创建该类的实例。

`wx.ClientDC` ：用于在一个窗口对象上绘画。当你想在窗口部件的主区域上（不包括边框或别的装饰）绘画时使用它。主区域有时也称为客户区。 `wx.ClientDC` 类也应临时创建。该类仅适用于 `wx.PaintEvent` 的处理之外。

`wx.MemoryDC` ：用于绘制图形到内存中的一个位图中，此时不被显示。然后你可以选择该位图，并使用 `wx.DC.Blit()` 方法来把这个位图绘画到一个窗口中。

`wx.MetafileDC` ：在 `Windows` 操作系统上， `wx.MetafileDC` 使你能够去创建标准窗口图元文件数据。

`wx.PaintDC`：等同于`wx.ClientDC`，除了它仅用于一个`wx.PaintEvent`的处理中。仅临时创建该类的实例。

`wx.PostScriptDC`：用于写压缩的`PostScript`文件。

`wx.PrinterDC`：用于`Windows`操作系统上，写到打印机。

`wx.ScreenDC`：用于直接在屏幕上绘画，在任何被显示的窗口的顶部或外部。该类只应该被临时创建。

`wx.WindowDC`：用于在一个窗口对象的整个区域上绘画，包括边框以及那些没有被包括在客户区域中的装饰。非`Windows`系统可能不支持该类。

下例6.1包含了显示图6.1的代码。因为该代码展示了基于设备上下文绘画的技巧，所以我们将对其详细注释。

例6.1 初始的`SketchWindow`代码

```
import wx

class SketchWindow(wx.Window):
    def __init__(self, parent, ID):
        wx.Window.__init__(self, parent, ID)
        self.SetBackgroundColour("White")
        self.color = "Black"
        self.thickness = 1
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)#
1 创建一个wx.Pen对象
        self.lines = []
        self.curLine = []
        self.pos = (0, 0)
        self.InitBuffer()

#2 连接事件
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)
        self.Bind(wx.EVT_LEFT_UP, self.OnLeftUp)
        self.Bind(wx.EVT_MOTION, self.OnMotion)
        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_IDLE, self.OnIdle)
        self.Bind(wx.EVT_PAINT, self.OnPaint)

    def InitBuffer(self):
        size = self.GetClientSize()

#3 创建一个缓存的设备上下文
        self.buffer = wx.EmptyBitmap(size.width, size.height)
        dc = wx.BufferedDC(None, self.buffer)

#4 使用设备上下文
        dc.SetBackground(wx.Brush(self.GetBackgroundColour()))
        dc.Clear()
        self.DrawLines(dc)
```

```
        self.reInitBuffer = False

    def GetLinesData(self):
        return self.lines[:]

    def SetLinesData(self, lines):
        self.lines = lines[:]
        self.InitBuffer()
        self.Refresh()

    def OnLeftDown(self, event):
        self.curLine = []
        self.pos = event.GetPositionTuple()#5 得到鼠标的位置
        self.CaptureMouse()#6 捕获鼠标

    def OnLeftUp(self, event):
        if self.HasCapture():
            self.lines.append((self.color,
                               self.thickness,
                               self.curLine))
            self.curLine = []
            self.ReleaseMouse()#7 释放鼠标

    def OnMotion(self, event):
        if event.Dragging() and event.LeftIsDown():#8 确定是否在拖
动
            dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)#9
 创建另一个缓存的上下文
            self.drawMotion(dc, event)
        event.Skip()
    #10 绘画到设备上下文
    def drawMotion(self, dc, event):
        dc.SetPen(self.pen)
        newPos = event.GetPositionTuple()
        coords = self.pos + newPos
        self.curLine.append(coords)
        dc.DrawLine(*coords)
        self.pos = newPos

    def OnSize(self, event):
        self.reInitBuffer = True #11 处理一个resize事件

    def OnIdle(self, event):#12 空闲时的处理
        if self.reInitBuffer:
            self.InitBuffer()
            self.Refresh(False)

    def OnPaint(self, event):
        dc = wx.BufferedPaintDC(self, self.buffer)#13 处理一个pai
nt（描绘）请求

    #14 绘制所有的线条
```

```
    def DrawLines(self, dc):
        for colour, thickness, line in self.lines:
            pen = wx.Pen(colour, thickness, wx.SOLID)
            dc.SetPen(pen)
            for coords in line:
                dc.DrawLine(*coords)

    def SetColor(self, color):
        self.color = color
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)

    def SetThickness(self, num):
        self.thickness = num
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

说明：

#1：`wx.Pen` 实例决定绘画到设备上下文的线条的颜色、粗细和样式。样式除了 `wx.SOLID` 还有 `wx.DOT`，`wx.LONGDASH`，和 `wx.SHORTDASH` 。

#2：窗口需要去响应几个不同的鼠标类型事件以便绘制图形。响应的事件有鼠标左键按下和释放、鼠标移动、窗口大小变化和窗口重绘。这里也指定了空闲时的处理。

#3：用两步创建了缓存的设备上下文：（1）创建空的位图，它作为画面外(`offscreen`)的缓存（2）使用画面外的缓存创建一个缓存的设备上下文。这个缓存的上下文用于防止我勾画线的重绘所引起的屏幕闪烁。在这节的较后面的部分，我们将更详细地讨论这个缓存的设备上下文。

#4：这几行发出绘制命令到设备上下文；具体就是，设置背景色并清空设备上下文(`dc.Clear()`)。必须调用 `dc.Clear()` ，其作用是产生一个 `wx.EVT_PAINT` 事件，这样，设置的背景就显示出来了，否则屏幕颜色不会改变。 `wx.Brush` 对象决定了背景的颜色和样式。

#5：事件方法 `GetPositionTuple()` 返回一个包含鼠标敲击的精确位置的 `Python` 元组。

#6： `CaptureMouse()` 方法控制了鼠标并在窗口的内部捕获鼠标，即使是你拖动鼠标到窗口边框的外面，它仍然只响应窗口内的鼠标动作。在程序的后面必须调用 `ReleaseMouse()` 来取消其对鼠标的控制。否则该窗口将无法通过鼠标关闭等，试将#7注释掉。

#7： `ReleaseMouse()` 方法将系统返回到调用 `CaptureMouse()` 之前的状态。 `wxPython` 应用程序使用一个椎栈来对捕获了鼠标的窗口的跟踪，调用 `ReleaseMouse()` 相当于从椎栈中弹出。这意味着你需要调用相同数据的 `CaptureMouse()` 和 `ReleaseMouse()` 。

#8：这行确定移动事件是否是线条绘制的一部分，由移动事件发生时鼠标左键是否处于按下状态来确定。 `Dragging()` 和 `LeftIsDown()` 都是 `wx.MouseEvent` 的方法，如果移动事件发生时所关联的条件成立，方法返回 `true` 。

#9：由于 `wx.BufferedDC` 是一个临时创建的设备上下文，所以在我们绘制线条之前需要另外创建一个。这里，我们创建一个新的 `wx.ClientDC` 作为主要的设备上下文，并再次使用我们的实例变量位图作为缓存。

#10：这几行实际是使用设备上下文去绘画新近的勾画线到屏幕上。首先，我们创建了 `coords` 元组，它合并了 `self.pos` 和 `newPos` 元组。这里，新的位置来自于事件 `GetPositionTuple()` ，老的位置是最后对 `OnMotion()` 调用所得到的。我们把该元组保存到 `self.curLine` 列表中，然后调用 `DrawLine()` 。
* `coords` 返回元组 `coords` 中的元素 `x1` , `y1` , `x2` , `y2` 。 `DrawLine()` 方法要求的参数形如 `x1` , `y1` , `x2` , `y2` ，并从点(`x1` , `y1`) 到(`x2` , `y2`) 绘制一条线。勾画的速度依赖于底层系统的速度。

#11：如果窗口大小改变了，我们存储一个 `True` 值到 `self.reInitBuffer` 实例属性中。我们实际上不做任何事直到下一个空闲事件。

#12：当一个空闲产生时，如果已发生了一个或多个尺寸改变事件，这个应用程序抓住时机去响应一个尺寸改变事件。我们存储一个 `True` 值到 `self.reInitBuffer` 实例属性中，并在一个空闲产生时响应的动机是避免对于接二连三的尺寸改变事件都进行屏幕刷新。

#13：对于所有的显示要求，都将产生 `wx.EVT_PAINT` 事件（描绘事件），并调用我们这里的方法 `OnPaint` 进行屏幕刷新（重绘），你可以看到这是出乎意料的简单：创建一个缓存的画图设备上下文。实际上 `wx.PaintDC` 被创建（因为我们处在一个 `Paint` 请求里，所以我们需要 `wx.PaintDC` 而非一个 `wx.ClientDC` 实例），然后在 `dc` 实例被删除后（函数返回时被销毁），位图被一块块地传送(`blit`) 给屏幕并最终显示。关于缓存的更详细的信息将在随后的段落中提供。

#14：当由于尺寸改变（和由于从文件载入）而导致应用程序需要根据实际数据重绘线条时，被使用。这里，我们遍历存储在实例变量 `self.lines` 中行的列表，为每行重新创建画笔，然后根据坐标绘制每一条线。

这个例子使用了两个特殊的 `wx.DC` 的子类，以使用绘画缓存。一个绘画缓存是一个不显现的区域，其中存储了所有的绘画命令（这些命令能够一次被执行），并且一步到位地复制到屏幕上。缓存的好处是用户看不到单个绘画命令的发生，因此屏

幕不会闪烁。正因如此，缓存被普遍地用于动画或绘制是由一些小的部分组成的场合。

在 `wxPython` 中，有两个用于缓存的类： `wx.BufferDC` （通常用于缓存一个 `wx.ClientDC` ）、 `wx.BufferPaintDC` （用于缓存一个 `wx.PaintDC` ）。它们工作方式基本上一样。缓存设备上下文的创建要使用两个参数。第一个是适当类型的目标设备上下文（例如，在例6.1中的#9，它是一个新的 `wx.ClientDC` 实例）。第二个是一个 `wx.Bitmap` 对象。在例6.1中，我们使用函数 `wx.EmptyBitmap` 创建一个位图。当绘画命令到缓存的设备上下文时，一个内在的 `wx.MemoryDC` 被用于位图绘制。当缓存对象被销毁时，C++销毁器使用 `Blit()` 方法去自动复制位图到目标。在 `wxPython` 中，销毁通常发生在对象退出作用域时。这意味缓存的设备上下文仅在临时创建时有用，所以它们能够被销毁并能用于块传送(`blit)` 。

例如例6.1的 `OnPaint` （）方法中， `self.buffer` 位图在建造勾画（ `sketch` ）期间已经被写了。只需要创建缓存对象，从而建立关于窗口的已有的位图与临时 `wx.PaintDC()` 之间的连接。方法结束后，缓存 `DC` 立即退出作用域，触发它的销毁器，同时将位图复制到屏幕。

设备上下文的函数

当你使用设备上下文时，要记住根据你的绘制类型去使用恰当的上下文（特别要记住 `wx.PaintDC` 和 `wx.ClientDC` 的区别）。一旦你有了适当的设备上下文，然后你就可以用它们来做一些事情了。表6.2列出 了 `wx.DC` 的一些方法。

表**6.2** `wx.DC` 的常用方法

`Blit(xdest` , `ydest` , `width` , `height` , `source` , `xsrc` , `ysrc)` ：从源设备上下文复制块到调用该方法的设备上下文。参数 `xdest` ， `ydest` 是复制到目标上下文的起始点。接下来的两个参数指定了要复制的区域的宽度和高度。 `source` 是源设备上下文， `xsrc` , `ysrc` 是源设备上下文中开始复制的起点。还有一些可选的参数来指定逻辑叠加功能和掩码。

`Clear()` ：通过使用当前的背景刷来清除设备上下文。

`DrawArc(x1` , `y1` , `x2` , `y2` , `xc` , `yc)` ：使用起点(`x1` , `y1)` 和终点(`x2` ， `y2)` 画一个圆弧。(`xc` ， `yc)` 是圆弧的中心。圆弧使用当前的画刷填充。这个函数按逆时针画。这也有一个相关的方法 `DrawEllipticalArc()` 。

`DrawBitmap(bitmap` , x,y, `transparent)` ：绘制一个 `wx.Bitmap` 对象，起点为(x, `y)` 。如果 `transparent` 为真，所复制的位图将是透明的。

`DrawCircle(x` , y, `radius)` `DrawCircle(point` , `radius)` ：按给定的中心点和半径画圆。这也有一个相关的方法 `DrawEllipse` 。

`DrawIcon(icon` , x, `y)` ：绘制一个 `wx.Icon` 对象到上下文，起点是(x, `y)` 。

`DrawLine(x1` , `y1` , `x2` , `y2)` ：从点(`x1` , `y1)` 到(`x2` , `y2)` 画一条线。这有一个相关的方法 `DrawLines()` ，该方法要 `wx.Point` 对象的一个 `Python` 列表为参数，并将其中的点连接起来。

`DrawPolygon(points)` ：按给定的 `wx.Point` 对象的一个 `Python` 列表绘制一个多边形。与 `DrawLines()` 不同的是，它的终点和起点相连。多边形使用当前的画刷来填充。这有一些可选的参数来设置x和y的偏移以及填充样式。

`DrawRectangle(x`, y, `width`, `height)` ：绘制一个矩形，它的左上角是(x, `y)` ，其宽和高是 `width` 和 `height`

。

`DrawText(text`, x, `y)` ：从点(x, `y)` 开始绘制给定的字符串，使用当前的字体。相关函数包括 `DrawRotatedText()` 和 `GetTextExtent()` 。文本项有前景色和背景色属性。

`FloodFill(x`, y, `color`, `style)` ：从点(x, `y)` 执行一个区域填充，使用当前画刷的颜色。参数 `style` 是可选的。 `style` 的默认值是 `wx.FLOOD_SURFACE` ，它表示当填充碰到另一颜色时停止。另一值 `wx.FLOOD_BORDER` 表示参数 `color` 是填充的边界，当填充碰到该颜色的代表的边界时停止。

`GetBackground()` `SetBackground(brush)` ：背景画刷是一个 `wx.Brush` 对象，当 `Clear()` 方法被调用时使用。

`GetBrush()` `SetBrush(brush)` ：画刷是一个 `wx.Brush` 对象并且用于填充任何绘制在设备上下文上的形状。

`GetFont()` `SetFont(font)` ：字体(`font)` 是一个 `wx.Font` 对象，被用于所有的文本绘制操作。

`GetPen()` `SetPen(pen)` ：画笔(`pen)` 是一个 `wx.Pen` 对象，被用于所有绘制线条的操作。

`GetPixel(x` , `y)` ：返回一个关于点(x, `y)` 的像素的一个 `wx.Colour` 对象。

`GetSize()` `GetSizeTuple()` ：以一个 `wx.Size` 对象或一个 `Python` 元组的形式返回设备上下文的像素尺寸。

上面的列表并没有囊括所有的方法。另外的一些方法将在第十二章中说明。

添加窗口装饰

尽管绘制到屏幕是一个画图程序不可或缺的部分，但是它距美观的程序还差的远。在这一节，我们将谈及常用的窗口装饰：状态栏、菜单和工具栏。我们将在第10章对这些做更详细的讨论。

如何添加和更新一个状态栏

在 `wxPython` 中，你可以通过调用框架的 `CreateStatusBar()` 方法添加并放置一个状态栏到一个框架的底部。当父框架调整大小的时候，状态栏自动的自我调整大小。默认情况下，状态栏是类 `wx.StatusBar` 的一个实例。要创建一个自定义

的状态栏，要使用 `SetStatusBar()` 方法并要求你的新类的实例作为参数来将状态栏附着到你的框架上。

要在你的状态栏上显示单一的一段文本，你可以使用 `wx.StatusBar` 的 `SetStatusText()` 方法。例6.2扩展了在例6.1中所演示的 `SketchFrame` 类来在状态栏中显示当前鼠标的位置。

例6.2 给框架添加一个简单的状态栏

```
# python

import wx from example1 import SketchWindow

class SketchFrame(wx.Frame):

    def init(self, parent):
        wx.Frame.init(self, parent, -1, "Sketch Frame",
            size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.statusbar = self.CreateStatusBar()

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText(str(event.GetPositionTuple(
)))
        event.Skip()

if name == 'main':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

我们通过使框架捕捉勾画窗的 `wx.EVT_MOTION` 事件来在状态栏中显示鼠标位置。事件处理器使用由该事件提供的数据设置状态栏的文?尽H缓蟮饔胷方法来保证另外的 `OnMotion()` 方法被调用，否则线条将不被绘制。

如果你想在状态栏中显示多个文本元素，你可以在状态栏中创建多个文本域。要使用这个功能，你要调用 `SetFieldsCount()` 方法，其参数是域的数量；默认情况下只有我们先前所见的那一个域。这之后使用先前的 `SetStatusText()` ，但是要使用第二个参数来指定此方法所应的域。域的编号从0开始。如果你不指定一个域，那么默认为设置第0号域，这也说明了为什么我们没有指定域而先前的例子能工作。

默认情况下，每个域的宽度是相同的。要调整文本域的尺寸， `wxPython` 提供了 `SetStatusWidth()` 方法。该方法要求一个整数的 `Python` 列表作为参数，列表的长度必须和状态栏中哉的数量一致。按列表中整数的顺序来计算对应域的宽度。如果整数是正值，那么宽度是固定的。如果你想域的宽度随框架的变化而变

化，那么应该使用负值。负值的绝对值代表域的相对宽度；可以把它认为是所占总宽度的比例。例如调用 `statusbar.SetStatusWidth(` [-1, -2,-3])方法所导致的各域从左到右的宽度比例是1:2:3。图6.2显示了这个结果。

图6.2



Figure 6.2 A sample status bar with the fields getting 1/6, 2/3, and 1/2 of the total width

例子6.3增加了两个状态域，其中一个显示所绘的当前线条的点数，另一个显示当前所画的线条的数量。该例所产生的状态条如图6.2所示。

例6.3 支持多个状态域

```
import wx
from example1 import SketchWindow

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetFieldsCount(3)
        self.statusbar.SetStatusWidths([-1, -2, -3])

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText("Pos: %s" %
                str(event.GetPositionTuple()), 0)
        self.statusbar.SetStatusText("Current Pts: %s" %
                len(self.sketch.curLine), 1)
        self.statusbar.SetStatusText("Line Count: %s" %
                len(self.sketch.lines), 2)
        event.Skip()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

`StatusBar` 类使你能够把状态域当作一个后进先出的堆栈。尽管本章的演示程序没有这样用， `PushStatus` - `Text()` 和 `PopStatusText()` 使得你能够在临时显示新的文本之后返回先前的状态文本。这两个方法都有一个可选的域号参数，以便在多个状态域的情况下使用。

表6.3归纳了 `wx.StatusBar` 最常用的方法

表6.3 **`wx.StatusBar`** 的方法

`GetFieldsCount()` `SetFieldsCount(count)` ：得到或设置状态栏中域的数量。

`GetStatusText(field` =0) `SetStatusText(text` , `field` =0)：得到或设置指定域中的文本。0是默认值，代表最左端的域。

`PopStatusText(field` =0)：弹出堆栈中的文本到指定域中，以改变域中的文本为弹出值。

`PushStatusText(text` , `field` =0)：改变指定的域中的文本为给定的文本，并将改变前的文本压入堆栈的顶部。

`SetStatusWidths(widths)` ：指定各状态域的宽度。 `widths` 是一个整数的 `Python` 列表。

在第10章中，我们将对状态栏作更详细的说明。下面我们将讨论菜单。

如何添加菜单？

本节，我们将说明如何添加子菜单和复选或单选菜单。子菜单是顶级菜单中的菜单。复制菜单或单选菜单是一组菜单项，它们的行为类似于一组复选框或单选按钮。图6.3显示了一个菜单栏，其中的一个子菜单包含了单选菜单项。

图6.3

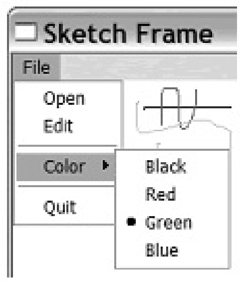


Figure 6.3
A menu that uses a submenu with radio menu items

要创建一个子菜单，首先和创建别的菜单方法一样创建一个菜单，然后再使用 `wx.Menu.AppendMenu()` 将它添加给父菜单。

带有复选或单选菜单的菜单可以通过使用 `wx.Menu` 的 `AppendCheckItem()` 和 `AppendRadioItem()` 方法来创建，或通过在 `wx.MenuItem` 的创建器中使参数 `kind` 的属性值为下列之一来创建： `wx.ITEM_NORMAL` , `wx.ITEM_CHECKBOX` , 或 `wx.ITEM_RADIO` 。要使用编程的方法来选择一个菜单项，可以使 `wx.Menu` 的 `Check(id` , `bool)` 方法， `id` 是所要改变项的 `wxPython` `ID` ， `bool` 指定了该项的选择状态。

例6.4为我们初始的绘画程序添加了菜单支持。我们这里的菜单改进自例5.5中的被重构的公用程序代码。

例子**6.4**

```
import wx
from example1 import SketchWindow

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar() #1 这里因重构有点变化
        self.createMenuBar()

    def initStatusBar(self):
        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetFieldsCount(3)
        self.statusbar.SetStatusWidths([-1, -2, -3])

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText("Pos: %s" %
                str(event.GetPositionTuple()), 0)
        self.statusbar.SetStatusText("Current Pts: %s" %
                len(self.sketch.curLine), 1)
        self.statusbar.SetStatusText("Line Count: %s" %
                len(self.sketch.lines), 2)
        event.Skip()

    def menuData(self): #2 菜单数据
        return [("  ", (
                    ("  ", "New Sketch file", self.OnNew),
                    ("  ", "Open sketch file", self.OnOpen),
                    ("  ", "Save sketch file", self.OnSave),
                    ("", "", ""),
                    ("  ", (
                        ("  ", "", self.OnColor,
                           wx.ITEM_RADIO),
                        ("  ", "", self.OnColor,
                           wx.ITEM_RADIO),
                        ("  ", "", self.OnColor,
                           wx.ITEM_RADIO),
                        ("  ", "", self.OnColor,
                           wx.ITEM_RADIO))),
                    ("", "", ""),
                    ("  ", "Quit", self.OnCloseWindow)))]

    def createMenuBar(self):
        menuBar = wx.MenuBar()
        for eachMenuData in self.menuData():
            menuLabel = eachMenuData[0]
            menuItems = eachMenuData[1]
            menuBar.Append(self.createMenu(menuItems), menuLabel
```

```
)
        self.SetMenuBar(menuBar)

    def createMenu(self, menuData):
        menu = wx.Menu()
#3 创建子菜单
        for eachItem in menuData:
            if len(eachItem) == 2:
                label = eachItem[0]
                subMenu = self.createMenu(eachItem[1])
                menu.AppendMenu(wx.NewId(), label, subMenu)

            else:
                self.createMenuItem(menu, *eachItem)
        return menu

    def createMenuItem(self, menu, label, status, handler,
                       kind=wx.ITEM_NORMAL):
        if not label:
            menu.AppendSeparator()
            return
        menuItem = menu.Append(-1, label, status, kind)#4 使用kin
d创建菜单项
        self.Bind(wx.EVT_MENU, handler, menuItem)

    def OnNew(self, event): pass
    def OnOpen(self, event): pass
    def OnSave(self, event): pass

    def OnColor(self, event):#5 处理颜色的改变
        menubar = self.GetMenuBar()
        itemId = event.GetId()
        item = menubar.FindItemById(itemId)
        color = item.GetLabel()
        self.sketch.SetColor(color)

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

说明：

#1：现在 `__init__` 方法包含了更多的功能，我们把状态栏放在了它自己的方法中。

#2：菜单数据的格式现在是(标签, (项目))，其中的每个项目也是一个列表(标签, 描术文字, 处理器, 可选的 `kind)` 或一个带有标签和项目的菜单。确定数据的一个子项目是菜单还是一个菜单项，请记住，菜单的长度是2，项目的长度是3或4。对于更复杂的产品数据，我建议使用 `XML` 或别的外部格式。

#3：如果数据块的长度是2，这意味它是一个菜单，将之分开，并递归调用 `createMenu` ，然后将之添加。

#4：创建菜单项。对 `wx.MenuItem` 的构造器使用 `kind` 参数的方法比使用 `wx.Menu` 的特定方法更容易。

#5： `OnColor` 方法根据所选菜单项来改变画笔的颜色。代码根据事件得到项目的 `id` ，再使用 `FindItemById()` 来得到正确的菜单项（注意我们这里使用菜单栏作为数据结构来访问，而没有使用项目 `id` 的哈希表），这个方法是以标签是 `wxPython` 颜色名为前提的。

如何添加一个工具栏

菜单栏和工具栏通常是紧密联系在一起的，工具栏的绝大部分功能与菜单项相对应。在 `wxPython` 中，这通过工具栏被敲击时发出 `wx.EVT_MENU` 事件，这样就可很容易地在处理菜单项的选择和工具栏的敲击时使用相同的方法。一个 `wxPython` 工具栏是类 `wx.ToolBar` 的一个实例，正如我们在第二章中所见的，可以使用框架的方法 `CreateToolBar()` 来创建。和状态栏一样，工具栏的大小也随其父框架的改变而自动改变。工具栏与其它的 `wxPython` 窗口一样可以拥有任意的子窗口。工具栏也包含创建工具按钮的方法。图6.4显示了带有一个工具栏的 `sketch` 窗口的一部分，这个工具栏使用了6.2.2中菜单的相应方法，以与菜单项的功能相对应。

图6.4

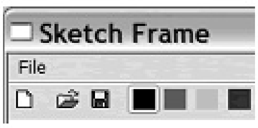


Figure 6.4 A typical toolbar showing both regular and toggle buttons

例6.5中，我们没有重复使用菜单改变画笔颜色的代码，而是使用了新的方法。

例**6.5** 添加一个工具栏到 **`sketch`** 应用程序

```
def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",size=
(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar()
        self.createMenuBar()
```

```
        self.createToolBar()

def createToolBar(self):#1创建工具栏
        toolbar = self.CreateToolBar()
        for each in self.toolbarData():
                self.createSimpleTool(toolbar, *each)
                toolbar.AddSeparator()
        for each in self.toolbarColorData():
                self.createColorTool(toolbar, each)
                toolbar.Realize()#2 显现工具栏

def createSimpleTool(self, toolbar, label, filename,help, handle
r):#3 创建常规工具
        if not label:
                toolbar.AddSeparator()
                return
        bmp = wx.Image(filename,wx.BITMAP_TYPE_BMP).ConvertToBit
map()
        tool = toolbar.AddSimpleTool(-1, bmp, label, help)
        self.Bind(wx.EVT_MENU, handler, tool)

def toolbarData(self):
        return (("New", "new.bmp", "Create new sketch",self.OnNe
w),
                ("", "", "", ""),
                ("Open", "open.bmp", "Open existing sketch",self
.OnOpen),
                ("Save", "save.bmp", "Save existing sketch",self
.OnSave))

def createColorTool(self, toolbar, color):#4 创建颜色工具
        bmp = self.MakeBitmap(color)
        newId = wx.NewId()
        tool = toolbar.AddRadioTool(-1, bmp, shortHelp=color)
        self.Bind(wx.EVT_MENU, self.OnColor, tool)

def MakeBitmap(self, color):#5 创建纯色的位图
        bmp = wx.EmptyBitmap(16, 15)
        dc = wx.MemoryDC()
        dc.SelectObject(bmp)
        dc.SetBackground(wx.Brush(color))
        dc.Clear()
        dc.SelectObject(wx.NullBitmap)
        return bmp

def toolbarColorData(self):
        return ("Black", "Red", "Green", "Blue")

def OnColor(self, event):#6 改变画笔颜色以响应工具栏的敲击
        menubar = self.GetMenuBar()
        itemId = event.GetId()
        item = menubar.FindItemById(itemId)
        if not item:
```

```
        toolbar = self.GetToolBar()
        item = toolbar.FindById(itemId)
        color = item.GetShortHelp()
        else:
        color = item.GetLabel()
        self.sketch.SetColor(color)
```

#1：工具栏的代码在设置上类似于菜单代码。然而，这里，我们对常规的按钮和单选切换按钮使用了不同的循环设置。

#2：`Realize()` 方法实际上是在工具栏中布局工具栏对象。它在工具栏被显示前必须被调用，如果工具栏中的添加或删除了工具，那么这个方法也必须被调用。

#3：这个方法类似于菜单项的创建。主要区别是工具栏上的工具要求显示位图。这里我们使用了三个位于同一目录下基本位图。在该方法的最后，我们绑定了菜单项所使用的相同的 `wx.EVT_MENU` 事件。

#4：颜色工具的创建类似于常规的工具。唯一的不同是使用了一个不同的方法去告诉工具栏它们是单选工具。纯色的位图由 `MakeBitmap()` 方法创建。

#5：该方法为单选工具创建纯色的位图。

#6：该方法在原有的基础上添加了搜索正确的工具以具此来改变颜色。然而，所写的代码的问题是，通过菜单项使画笔颜色改变了，但是工具栏上的单选工具的状态没有相应改变，反过来也是一样。

工具栏中的工具在鼠标右键敲击时能够产生 `wx.EVT_TOOL_RCLICKED` 类型事件。工具栏也有一些不同的样式，它们被作为位图参数传递给 `CreateToolBar()` 。表6.4列出了一些工具栏的样式。

表**6.4** **`wx.ToolBar`** 类的样式

`wx.TB_3DBUTTONS` ：3D外观

`wx.TB_HORIZONTAL` ：默认样式，工具栏水平布置

`wx.TB_NOICONS` ：不为每个工具显示位图

`wx.TB_TEXT` ：根据不同的位图显示简短的帮助文本

`wx.TB_VERTICAL` ：垂直放置工具栏

工具栏比状态栏更复杂。表6.5显示了其常用的一些方法。

表**6.5** **`wx.ToolBar`** 的常用方法

`AddControl(control)` ：添加一个任意的 `wxPython` 控件到工具栏。相关方法 `InsertControl()` 。

`AddSeparator()` ：在工具之间放置空格。

`AddSimpleTool(id`, `bitmap`, `shortHelpString` ="", `kind` = `wx.ITEM_NORMAL)`：添加一个简单的带有给定位图的工具到工具栏。`shortHelpString` 作为提示显示。`kind` 的值可以是 `wx.ITEM_NORMAL`，`wx.ITEM_CHECKBOX`，或 `wx.ITEM_RADIO`。

`AddTool(id`, `bitmap`, `bitmap2` = `wx.NullBitmap`, `kind` = `wx.ITEM_NORMAL`, `shortHelpS` ="", `longHelpString` ="", `clientData` = `None)`：简单工具的其它参数。`bitmap2` 是当该工具被按下时所显示的位图。`longHelpString` 是当指针位于该工具中时显示在状态栏中的帮助字符串。`clientData` 用于将任意的一块数据与工具相联系起来。相关方法 `InsertTool()`。

`AddCheckTool(...)`：添加一个复选框切换工具，所要求参数同 `AddTool()`。

`AddRadioTool(...)`：添加一个单选切换工具，所要求参数同 `AddTool()`。对于连续的未分隔的单选工具被视为一组。

`DeleteTool(toolId)` `DeleteToolByPosition(x`, `y)`：删除所给定的 `id` 的工具，或删除给定显示位置的工具。

`FindControl(toolId)` `FindToolForPosition(x`, `y)`：查找并返回给定 `id` 或显示位置的工具。

`ToggleTool(toolId`, `toggle)`：根据布尔什 `toggle` 来设置给定 `id` 的工具的状态。

下一节，我们将给你展示如何使用通用对话框来得到用户的信息。

得到标准信息

你的应用程序经常需要从用户那里得到基本的信息，这通常通过对话框。在这一节，我们将讨论针对标准用户信息的标准文件和颜色对话框。

如何使用标准文件对话框？

大部分的 `GUI` 应用程序都要保存和载入这样那样的数据。考虑到你和你的用户，应该有一个简单的，方便的机制来选择文件。很高兴，为此 `wxPython` 提供了标准的文件对话框 `wx.FileDialog`。图6.5显示了这个用于 `sketch` 程序的文件对话框。

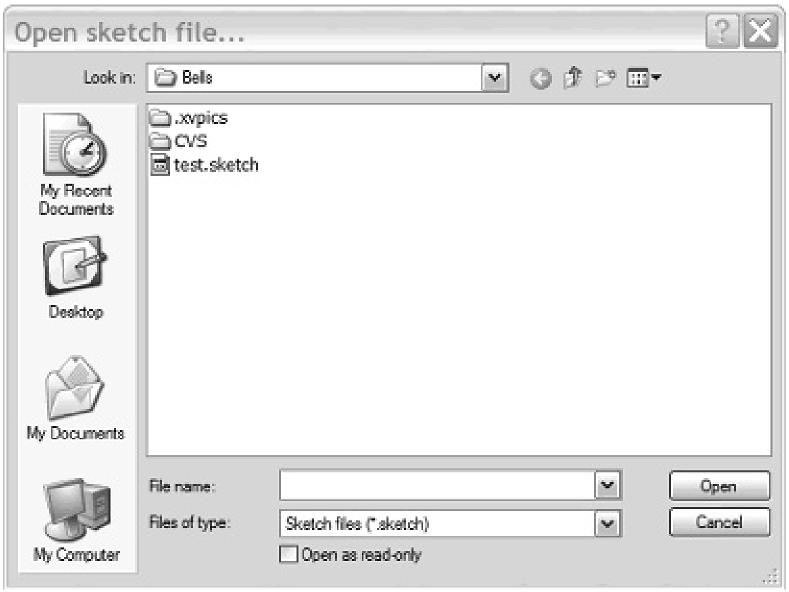


Figure 6.5
A standard file dialog for Windows

`wx.FileDialog`最重要的方法是它的构造器，语法如下：

`wx.FileDialog(parent` , `message` =" `Choose` a `file` ", `defaultDir` ="", `defaultFile` ="", `wildcard` =".", `style` =0)

表6.6对构造器的参数进行了说明。

表**6.6** **`wx.FileDialog`** 构造器的参数

`parent` ：对话框的父窗口。如果没有父窗口则为 `None` 。

`message` ： `message` 显示在对话框的标题栏中。

`defaultDir` ：当对话框打开时，默认的目录。如果为空，则为当前工作目录。

`defaultFile` ：当对话框打开时，默认选择的文件。如果为空，则没有文件被选择。

`wildcard` ：通配符。指定要选择的文件类型。格式是 `display` | `wildcard` 。可以指定多种类型的文件，例如：“ `Sketch` `files` (. *`sketch)`* |. `sketch` | `All` `files` (.)|.”。

`style` ：样式。见下表6.7。

表**6.7** **`wx.FileDialog`** 的样式

`wx.CHANGE_DIR` ：在用户选择了一个文件之后，当前工作目录相应改变到所选文件所在的目录。

`wx.MULTIPLE` ：仅适用于打开对话框。这个样式使得用户可以选择多个文件。

`wx.OPEN` ：这个样式用于打开一个文件。

`wx.OVERWRITE_PROMPT` ：仅适用于保存文件对话框。显示一个提示信息以确认是否覆盖一个已存在的文件。

`wx.SAVE` ：这个样式用于保存一个文件。

要使用文件对话框，要对一个对话框实例调用 `ShowModal()` 方法。这个方法根据用户所敲击的对话框上的按钮来返回 `wx.ID_OK` 或 `wx.ID_CANCEL` 。选择之后。使用 `GetFilename()` , `GetDirectory()` ,或 `GetPath()` 方法来获取数据。之后，调用 `Destroy()` 销毁对话框是一个好的观念。

下例6.6显了对 `SketchFrame` 所作的修改以提供保存和装载（完整的附书源码请到论坛的"相关资源"的"教程下载"中下载）。这些改变要求导入 `cPickle` 和 `os` 模块。我们使用 `cPickle` 来将数据的列表转换为可用于文件读写的数据格式。

例**6.6** **`SketchFrame`** 的保存和装载方法

```
def __init__(self, parent):
        self.title = "Sketch Frame"
        wx.Frame.__init__(self, parent, -1, self.title,size=(800
,600))
        self.filename = ""
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar()
        self.createMenuBar()
        self.createToolBar()

def SaveFile(self):#1 保存文件
        if self.filename:
                data = self.sketch.GetLinesData()
                f = open(self.filename, 'w')
                cPickle.dump(data, f)
                f.close()

def ReadFile(self):#2 读文件
        if self.filename:
                try:
                        f = open(self.filename, 'r')
                        data = cPickle.load(f)
                        f.close()
                        self.sketch.SetLinesData(data)
                except cPickle.UnpicklingError:
                        wx.MessageBox("%s is not a sketch file."
 % self.filename, "oops!",
                        style=wx.OK|wx.ICON_EXCLAMATION)

        wildcard = "Sketch files (*.sketch)|*.sketch|All files (
*.*)|*.*"

def OnOpen(self, event):#3 弹出打开对话框
        dlg = wx.FileDialog(self, "Open sketch file...",os.getcw
d(), style=wx.OPEN,wildcard=self.wildcard)
        if dlg.ShowModal() == wx.ID_OK:
                self.filename = dlg.GetPath()
                self.ReadFile()
```

```
                self.SetTitle(self.title + ' -- ' + self.filenam
e)
                dlg.Destroy()

def OnSave(self, event):#4 保存文件
        if not self.filename:
                self.OnSaveAs(event)
        else:
                self.SaveFile()

def OnSaveAs(self, event):#5 弹出保存对话框
        dlg = wx.FileDialog(self, "Save sketch as...",
        os.getcwd(),
        style=wx.SAVE | wx.OVERWRITE_PROMPT,
        wildcard=self.wildcard)
        if dlg.ShowModal() == wx.ID_OK:
                filename = dlg.GetPath()
        if not os.path.splitext(filename)[1]:#6 确保文件名后缀
                filename = filename + '.sketch'
                self.filename = filename
                self.SaveFile()
                self.SetTitle(self.title + ' -- ' +
                self.filename)
        dlg.Destroy()
```

#1：该方法写文件数据到磁盘中，给定了文件名，使用了 `cPickle` 模块。

#2：该方法使用 `cPickle` 来读文件。如果文件不是期望的类型，则弹出一个消息框来警告。

#3： `OnOpen()` 方法使用 `wx.OPEN` 样式来创建一个对话框。通配符让用户可以限定选择. `sketch` 文件。如果用户敲击 `OK` ，那么该方法根据所选择的路径调用 `ReadFile()` 方法。

#4：如果已经选择了用于保存当前数据的文件名，那么保存文件，否则，我们打开保存对话框。

#5： `OnSave()` 方法创建一个 `wx.SAVE` 文件对话框。

#6：这行确保文件名后缀为. `sketch` 。

下一节，我们将讨论如何使用文件选择器。

如何使用标准的颜色选择器？

如果用户能够在 `sketch` 对话框中选择任意的颜色，那么这将是有用。对于这个目的，我们可以使用 `wxPython` 提供的标准 `wx.ColourDialog` 。这个对话框的用法类似于文件对话框。它的构造器只需要一个 `parent(` 双亲)和一个可选的数据属性

参数。数据属性是一个 `wx.ColourData` 的实例，它存储与该对话框相关的一些数据，如用户选择的颜色，还有自定义的颜色的列表。使用数据属性使你能够在以后的应用中保持自定义颜色的一致性。

在 `sketch` 应用程序中使用颜色对话框，要求增加一个菜单项和一个处理器方法。例6.7显示了所增加的代码。

例**6.7** 对 **`SketchFrame`** 做一些改变，以显示颜色对话框

```
def menuData(self):
        return [("  ", (
                ("  ", "New Sketch file", self.OnNew),
                ("  ", "Open sketch file", self.OnOpen),
                ("  ", "Save sketch file", self.OnSave),
                ("", "", ""),
                ("  ", (
                ("  ", "", self.OnColor,wx.ITEM_RADIO),
                ("  ", "", self.OnColor,wx.ITEM_RADIO),
                ("  ", "", self.OnColor,wx.ITEM_RADIO),
                ("  ", "", self.OnColor,wx.ITEM_RADIO),
                ("  ", "", self.OnOtherColor,wx.ITEM_RADIO))),
                ("", "", ""),
                ("  ", "Quit", self.OnCloseWindow)))]

def OnOtherColor(self, event):
        dlg = wx.ColourDialog(self)
        dlg.GetColourData().SetChooseFull(True)# 创建颜色数据对象
        if dlg.ShowModal() == wx.ID_OK:
                self.sketch.SetColor(dlg.GetColourData().GetColo
ur())# 根据用户的输入设置颜色
        dlg.Destroy()
```

颜色数据实例的 `SetChooseFull()` 方法告诉对话框去显示整个调色板，其中包括了自定义的颜色信息。对话框关闭后，我们根据得到的颜色来拾取颜色数据。颜色数据作为一个 `wx.Color` 的实例返回并传递给 `sketch` 程序来设置颜色。

给应用程序一个好看的外观

在这一节中，我们将讨论如何让你的程序有一个好的外观。从重要性来说，储如你如何作安排以便用户调整窗口的大小，从细节来说，储如你如何显示一个 `about` 框。在本书的第二部分，我们将更详细地对这些主题进行进一步的讨论。

如何布局窗口部件？

在你的 `wxPython` 应用程序中布局你的窗口部件的方法之一是，在每个窗口部件被创建时显式地指定它的位置和大小。虽然这个方法相当地简单，但是它一直存在几个缺点。其中之一就是，因为窗口部件的尺寸和默认字体的尺寸不同，对于在所有

系统上要得到一个正确的定位是非常困难的。另外，每当用户调整父窗口的大小时，你必须显式地改变每个窗口部件的定位，要正确地实现它是十分痛苦的。

幸运的是，这儿有一个更好的方法。在 `wxPython` 中的布局机制是一个被称为 `sizer` 的东西，它类似于 `Java` `AWT` 和其它的界面工具包中的布局管理器。每个不同的 `sizer` 基于一套规则管理它的窗口的尺寸和位置。 `sizer` 属于一个容器窗口（比如 `wx.Panel` ）。在父中创建的子窗口必须被添加给 `sizer` ， `sizer` 管理每个窗口部件的尺寸和位置。

创建一个 **`sizer`**

创建一个 `sizer` 的步骤：

1、创建你想用来自动调用尺寸的 `panel` 或 `container(` 容器)。 2、创建 `sizer` 。 3、创建你的子窗口。 4、使用 `sizer` 的 `Add()` 方法来将每个子窗口添加给 `sizer` 。当你添加窗口时，给了 `sizer` 附加的信息，这包括窗口周围空间的度量、在由 `sizer` 所管理分配的空间中如何对齐窗口、当容器窗口改变大小时如何扩展子窗口等。 5、 `sizer` 可以嵌套，这意味你可以像窗口对象一样添加别的 `sizer` 到父 `sizer` 。你也可以预留一定数量的空间作为分隔。 6、调用容器的 `SetSizer(sizer)` 方法。

表6.8列出了在 `wxPython` 中有效的最常用的 `sizer` 。对于每个专门的 `sizer` 的更完整的说明见第11章。

表**6.8** 最常用的 **`wxPython`** 的 **`sizer`**

`wx.BoxSizer` ：在一条线上布局子窗口部件。 `wx.BoxSizer` 的布局方向可以是水平或竖直的，并且可以在水平或竖直方向上包含子 `sizer` 以创建复杂的布局。在项目被添加时传递给 `sizer` 的参数控制子窗口部件如何根据 `box` 的主体或垂直轴线作相应的尺寸调整。

`wx.FlexGridSizer` ：一个固定的二维网格，它与 `wx.GridSizer` 的区别是，行和列根据所在行或列的最大元素分别被设置。

`wx.GridSizer` ：一个固定的二维网格，其中的每个元素都有相同的尺寸。当创建一个 `grid` `sizer` 时，你要么固定行的数量，要么固定列的数量。项目被从左到右的添加，直到一行被填满，然后从下一行开始。

`wx.GridBagSizer` ：一个固定的二维网格，基于 `wx.FlexGridSizer` 。允许项目被放置在网格上的特定点，也允许项目跨越多和网格区域。

`wx.StaticBoxSizer` ：等同于 `wx.BoxSizer` ，只是在 `box` 周围多了一个附加的边框（有一个可选的标签）。

使用 **`sizer`**

为了演示 `sizer` 的用法，我们将给 `sketch` 应用程序增加一个 `control` `panel` 。 `control` `panel` 包含用来设置线条颜色和粗细的按钮。这个例子使用了 `wx.GridSizer` （用于按钮）和 `wx.BoxSizer` （用于其余的布局部分）。图6.6显示了使用了 `panel` 的 `sketch` 应用程序，并图解了 `grid` 和 `box` 的实际布局。



Figure 6.6 The Sketch application with an automatically laid out control panel

例6.8显示了实现 `control panel` 而对 `sketch` 程序所作的必要的改变。在这一节，我们的讨论将着重于 `sizer` 的实现。

例**6.8**

```
def __init__(self, parent):
        self.title = "Sketch Frame"
        wx.Frame.__init__(self, parent, -1, self.title, size=(80
0,600))
        self.filename = ""
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar()
        self.createMenuBar()
        self.createToolBar()
        self.createPanel()

def createPanel(self):
        controlPanel = ControlPanel(self, -1, self.sketch)
        box = wx.BoxSizer(wx.HORIZONTAL)
        box.Add(controlPanel, 0, wx.EXPAND)
        box.Add(self.sketch, 1, wx.EXPAND)
        self.SetSizer(box)
```

在例6.8中， `createPanel()` 方法创建了 `ControlPanel` （在下面的列表中说明）的实例，并且与 `box sizer` 放在一起。 `wx.BoxSizer` 的构造器的唯一参数是方向，取值可以是 `wx.HORIZONTAL` 或 `wx.VERTICAL` 。接下来，这个新

的 `controlPanel` 和先前创建的 `SketchWindow` 被使用 `Add()` 方法添加给了 `sizer` 。第一个参数是要被添加给 `sizer` 的对象。第二个参数是被 `wx.BoxSizer` 用作因数去决定当 `sizer` 的大小改变时， `sizer` 应该如何调整它的孩子的尺寸。我们这里使用的是水平方向调整的 `sizer` ， `stretch` 因数决定每个孩子的水平尺寸如何改变（坚直方向的改变由 `box` `sizer` 基于第三个参数来决定）。

如果第二个参数（ `stretch` 因数）是0，对象将不改变尺寸，无论 `sizer` 如何变化。如果第二个参数大于0，则 `sizer` 中的孩子根据因数分割 `sizer` 的总尺寸（类似于 `wx.StatusBar` 管理文本域的宽度的做法）。如果 `sizer` 中的所有孩子有相同的因数，那么它们按相同的比例分享放置了固定尺寸的元素后剩下的空间。这里的0表示假如用户伸展框架时， `controlPanel` 不改变水平的尺寸，而1表示绘画窗口（ `sketch` `window` ）的尺寸要随框架的改变而改变。

`Add()` 的第三个参数是另一个位掩码标志。完整的说明将在以后的章节中给出。 `wx.EXPAND` 指示 `sizer` 调整孩子的大小以完全填满有效的空间。其它的可能的选项允许孩子被按比例的调整尺寸或根据 `sizer` 的特定部分对齐。图6.7将帮助阐明参数及其控制的调整尺寸的方向。

这些设置的结果是当你运行这个带有 `box` `sizer` 的框架的时候，任何在水平方向的改变都将导致 `sketch` `window` 的尺寸在该方向?系母谋洌缂 `panel` 不会在该方向上改变。在坚直方向的尺寸改变导致这两个子窗口都要在坚直方向缩放。

例6.8中涉及的类 `ControlPanel` 结合使用了 `grid` 和 `box` `sizer` 。例6.9包含了这个类的代码。

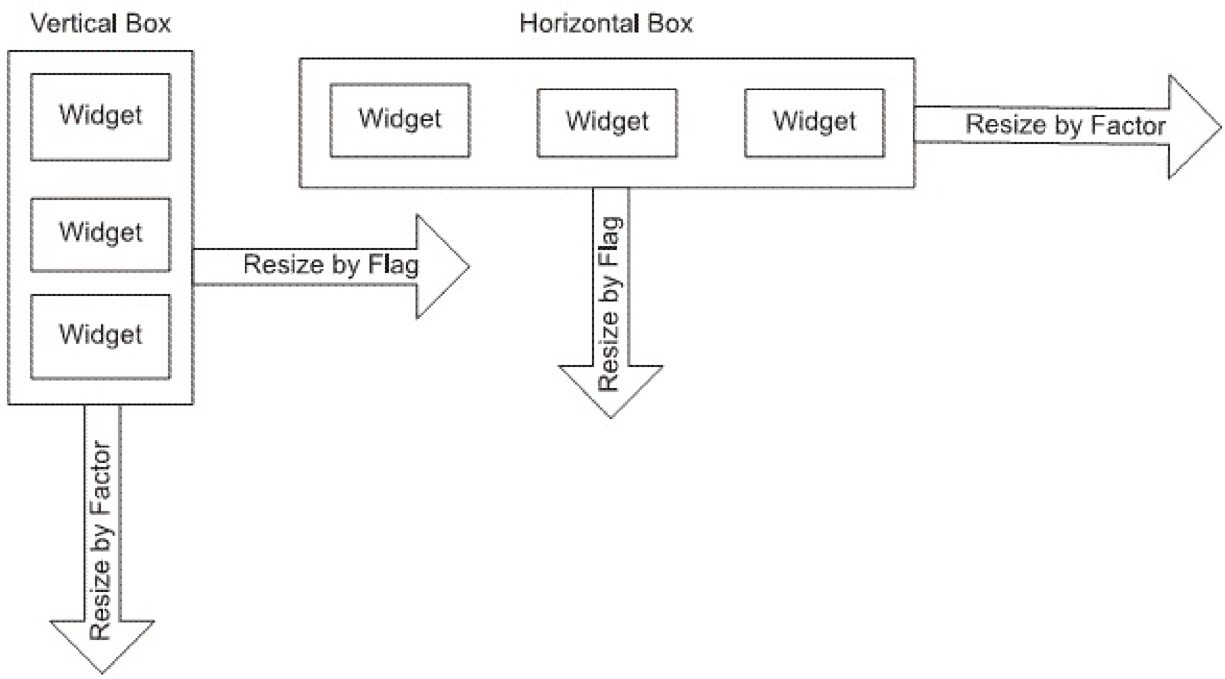


Figure 6.7 A drawing showing which argument determines resize behavior in each direction.

例**6.9** **`ControlPanel`** 类

```
class ControlPanel(wx.Panel):
```

```
    BMP_SIZE = 16
    BMP_BORDER = 3
    NUM_COLS = 4
    SPACING = 4

    colorList = ('Black', 'Yellow', 'Red', 'Green', 'Blue', 'Pur
ple',
              'Brown', 'Aquamarine', 'Forest Green', 'Light Blue
',
              'Goldenrod', 'Cyan', 'Orange', 'Navy', 'Dark Grey'
,
              'Light Grey')
    maxThickness = 16

    def __init__(self, parent, ID, sketch):
        wx.Panel.__init__(self, parent, ID, style=wx.RAISED_BORD
ER)
        self.sketch = sketch
        buttonSize = (self.BMP_SIZE + 2 * self.BMP_BORDER,
                      self.BMP_SIZE + 2 * self.BMP_BORDER)
        colorGrid = self.createColorGrid(parent, buttonSize)
        thicknessGrid = self.createThicknessGrid(buttonSize)
        self.layout(colorGrid, thicknessGrid)

    def createColorGrid(self, parent, buttonSize):#1 创建颜色网格
        self.colorMap = {}
        self.colorButtons = {}
        colorGrid = wx.GridSizer(cols=self.NUM_COLS, hgap=2, vga
p=2)
        for eachColor in self.colorList:
            bmp = parent.MakeBitmap(eachColor)
            b = buttons.GenBitmapToggleButton(self, -1, bmp, siz
e=buttonSize)
            b.SetBezelWidth(1)
            b.SetUseFocusIndicator(False)
            self.Bind(wx.EVT_BUTTON, self.OnSetColour, b)
            colorGrid.Add(b, 0)
            self.colorMap[b.GetId()] = eachColor
            self.colorButtons[eachColor] = b
        self.colorButtons[self.colorList[0]].SetToggle(True)
        return colorGrid

    def createThicknessGrid(self, buttonSize):#2 创建线条粗细网格
        self.thicknessIdMap = {}
        self.thicknessButtons = {}
        thicknessGrid = wx.GridSizer(cols=self.NUM_COLS, hgap=2,
 vgap=2)
        for x in range(1, self.maxThickness + 1):
            b = buttons.GenToggleButton(self, -1, str(x), size=b
uttonSize)
            b.SetBezelWidth(1)
            b.SetUseFocusIndicator(False)
```

```
                self.Bind(wx.EVT_BUTTON, self.OnSetThickness, b)
                thicknessGrid.Add(b, 0)
                self.thicknessIdMap[b.GetId()] = x
                self.thicknessButtons[x] = b
            self.thicknessButtons[1].SetToggle(True)
            return thicknessGrid

        def layout(self, colorGrid, thicknessGrid):#3 合并网格
            box = wx.BoxSizer(wx.VERTICAL)
            box.Add(colorGrid, 0, wx.ALL, self.SPACING)
            box.Add(thicknessGrid, 0, wx.ALL, self.SPACING)
            self.SetSizer(box)
            box.Fit(self)

        def OnSetColour(self, event):
            color = self.colorMap[event.GetId()]
            if color != self.sketch.color:
                self.colorButtons[self.sketch.color].SetToggle(False
)
            self.sketch.SetColor(color)

        def OnSetThickness(self, event):
            thickness = self.thicknessIdMap[event.GetId()]
            if thickness != self.sketch.thickness:
                self.thicknessButtons[self.sketch.thickness].SetTogg
le(False)
            self.sketch.SetThickness(thickness)
```

#1： `createColorGrid()` 方法建造包含颜色按钮的 `grid sizer` 。首先，我们创建 `sizer` 本身，指定列为4列。由于列数已被设定，所以按钮将被从左到右的布局，然后向下。接下来我们要求颜色的列表，并为每种颜色创建一个按钮。在 `for` 循环中，我们为每种颜色创建了一个方形的位图，并使用 `wxPython` 库中所定义的一般的按钮窗口部件类创建了带有位图的切换按钮。然后我们把按钮与事件相绑定，并把它添加到 `grid` 。之后，我们把它添加到字典以便在以后的代码中，易于关联颜色、 `ID` 和按钮。我们不必指定按钮在网格中的位置； `sizer` 将为我们做这件事。

#2： `createThicknessGrid()` 方法基本上类似于 `createColorGrid()` 方法。实际上，一个有进取心的程序员可以把它们做成一个通用函数。 `grid sizer` 被创建，十六个按钮被一次性添加， `sizer` 确保了它们在屏幕上很好地排列。

#3：我们使用一个竖直的 `box sizer` 来放置网格(`grid`) 。每个 `grid` 的第二个参数都是0，这表明 `grid sizer` 当 `control panel` 在垂直方向伸展时不改变尺寸。（由于我们已经知道 `control panel` 不在水平方向改变尺寸，所以我们不必指定水平方向的行为。） `Add()` 的第四个参数是项目的边框宽度，这里使用 `self.SPACING` 变量指定。第三个参数 `wx.ALL` 是一套标志中的一个，它控制那些边套用第四个参数指定的边框宽度， `wx.ALL` 表明对象的四个边都套用。最后，我们调用 `box sizer` 的 `Fit()` 方法，使用的参数是 `control panel` 。

这个方法告诉 `control` `panel` 调整自身尺寸以匹配 `sizer` 认为所需要的最小化尺寸。通常这个方法在使用了 `sizer` 的窗口的构造中被调用，以确保窗口的大小足以包含 `sizer` 。

基类 `wx.Sizer` 包含了几个通用于所有 `sizer` 的方法。表6.9列出了最常用的方法。

表6.9 `wx.Sizer` 的方法

```
Add(window, proportion=0,
        flag=0, border=0,
        userData=None)

Add(sizer, proportion=0,
        flag=0, border=0,
        userData=None)
```

`Add(size` , `proportion` =0, `flag` =0, `border` =0, `userData` = `None)` ：第一个添加一个 `wxWindow` ，第二个添加一个嵌套的 `sizer` ，第三个添加空的空间，用作分隔符。参数 `proportion` 管理窗口总尺寸，它是相对于别的窗口的改变而言的，它只对 `wx.BoxSizer` 有意义。参数 `flag` 是一个位图，针对对齐、边框位置，增长有许多不同的标志，完整的列表见第十一章。参数 `border` 是窗口或 `sizer` 周围以像素为单位的空间总量。 `userData` 使你能够将对象与数据关联，例如，在一个子类中，可能需要更多的用于尺寸的信息。

`Fit(window)` `FitInside(window`)：调整 `window` 尺寸以匹配 `sizer` 认为所需要的最小化尺寸。这个参数的值通常是使用 `sizer` 的窗口。 `FitInside()` 是一个类似的方法，只不过将改变窗口在屏幕上的显示替换为只改变它的内部实现。它用于 `scroll` `panel` 中的窗口以触发滚动栏的显示。

`GetSize()` ：以 `wx.Size` 对象的形式返回 `sizer` 的尺寸。

`GetPosition()` ：以 `wx.Point` 对象的形式返回 `sizer` 的位置。

`GetMinSize()` ：以 `wx.Size` 对象的形式返回完全填充 `sizer` 所需的最小尺寸。

`Layout()` ：强迫 `sizer` 去重新计算它的孩子的尺寸和位置。在动态地添加或删除了一个孩子之后调用。

`Prepend(...)` ：与 `Add()` 相同（只是为了布局的目的，把新的对象放在 `sizer` 列表的开头）。

`Remove(window)` `Remove(sizer)` `Remove(nth)` ：从 `sizer` 中删除一个对象。

`SetDimension(x` , y, `width` , `height)` ：强迫 `sizer` 按照给定的参数重新定位它的所有孩子。

有关 `sizer` 和嵌套 `sizer` 的更详细的信息请参考第11章。

如何建造一个关于(about)框？

`about` 框是显示对话框的一个好的例子，它能够显示比纯信息框更复杂的信息。这里，你可以使用 `wx.html.HtmlWindow` 作为一个简单的机制来显示样式文本。实际上，`wx.html.HtmlWindow` 远比我们这里演示的强大，它包括了管理用户交互以及绘制的方法。第16章涵盖了 `wx.html.HtmlWindow` 的特性。例6.10展示了一个类，它使用 `HTML` `renderer` 创建一个 `about` 框。

例**6.10** 使用 **`wx.html.HtmlWindow`** 作为一个 **`about`** 框

```
class SketchAbout(wx.Dialog):
    text = '''
<html>
<body bgcolor="#ACAA60">
<center><table bgcolor="#455481" width="100%" cellspacing="0"
cellpadding="0" border="1">
<tr>
<td align="center"><h1>Sketch!</h1></td>
</tr>
</table>
</center>
<p><b>Sketch</b> is a demonstration program for
<b>wxPython In Action</b>
Chapter 6\. It is based on the SuperDoodle demo included
with wxPython, available at http://www.wxpython.org/
</p>
<p><b>SuperDoodle</b> and <b>wxPython</b> are brought to you by
<b>Robin Dunn</b> and <b>Total Control Software</b>, Copyright
&copy; 1997-2006.</p>
</body>
</html>s
'''

    def __init__(self, parent):
        wx.Dialog.__init__(self, parent, -1, 'About Sketch',
                          size=(440, 400) )

        html = wx.html.HtmlWindow(self)
        html.SetPage(self.text)
        button = wx.Button(self, wx.ID_OK, "Okay")

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(html, 1, wx.EXPAND|wx.ALL, 5)
        sizer.Add(button, 0, wx.ALIGN_CENTER|wx.ALL, 5)

        self.SetSizer(sizer)
        self.Layout()
```

上面的 HTML 字符串中，有一些布局和字体标记。这里的对话框合并了 wx.html.HtmlWindow 和一个 wx.ID_OK ID 按钮。敲击按钮则自动关闭窗口，如同其它对话框一样。一个垂直的 box sizer 用于管理这个布局。

图6.8显示了该对话框。 图6.8



Figure 6.8
The HTML about box

把它作为一个菜单项(About) 的处理器的方法如下：

```
def OnAbout(self, event):
        dlg = SketchAbout(self)
        dlg.ShowModal()
        dlg.Destroy()
```

如何建造一个启动画面？

显示一个好的启动画面，将给你的用户一种专业化的感觉。在你的应用程序完成一个费时的设置的时候，它也可以转移用户的注意力。在 wxPython 中，使用类 wx.SplashScreen 建造一个启动画面是很容易的。启动画面可以保持显示指定的时间，并且无论时间是否被设置，当用户在其上敲击时，它总是会关闭。

wx.SplashScreen 类的构造函数如下：

```
wx.SplashScreen(bitmap, splashStyle, milliseconds, parent, id,
        pos=wx.DefaultPosition, size=wx.DefaultSize,
        style=wx.SIMPLE_BORDER|wx.FRAME_NO_TASKBAR|wx.STAY_ON_TO
P)
```

表6.10说明了 wx.SplashScreen 构造函数的参数

表6.10 wx.SplashScreen 构造函数的参数

`bitmap` ：一个 `wx.Bitmap` ，它被显示在屏幕上。

`splashStyle` ：另一个位图样式，可以是下列的结合： `wx.SPLASH_CENTRE_ON_PARENT` , `wx.SPLASH_CENTRE_ON_SCREEN` , `wx.SPLASH_NO_CENTRE` , `wx.SPLASH_TIMEOUT` , `wx.SPLASH_NO_TIMEOUT`

`milliseconds` ：如果 `splashStyle` 指定为 `wx.SPLASH_TIMEOUT` ， `milliseconds` 是保持显示的毫秒数。

`parent` ：父窗口，通常为 `None` 。

`id` ：窗口 `ID` ，通常使用-1比较好。

`pos` ：如果 `splashStyle` 取值 `wx.SPLASH_NO_CENTER` 的话， `pos` 指定画面在屏幕上的位置。

`size` ：尺寸。通常你不需要指定这个参数，因为通常使用位图的尺寸。

`style` ：常规的 `wxPython` 框架的样式，一般使用默认值就可以了。

例6.11显示了启动画面的代码。这里我们用一具自定义的 `wx.App` 子类来替代了 `wx.PySimpleApp` 。

例**6.11** 一个启动画面的代码

```
class SketchApp(wx.App):

    def OnInit(self):
        bmp = wx.Image("splash.png").ConvertToBitmap()
        wx.SplashScreen(bmp, wx.SPLASH_CENTRE_ON_SCREEN | wx.SPL
ASH_TIMEOUT,
                1000, None, -1)
        wx.Yield()

        frame = SketchFrame(None)
        frame.Show(True)
        self.SetTopWindow(frame)
        return True
```

通常，启动画面被声明在应用程序启动期间的 `OnInit` 方法中。启动画面将一直显示直到它被敲击或超时。这里，启动画面显示在屏幕的中央，一秒后超时。 `Yield()` 的调用很重要，因为它使得在应用程序继续启动前，任何未被处理的事件仍可以被继续处理。这里， `Yield()` 的调用确保了在应用程序继续启动前，启动画面能够接受并处理它的初始化绘制事件。

本章小结

1. 大多数的应用程序使用了诸如菜单、工具栏和启动画面这样的通常的元素。它们的使用不但对你程序的可用性有帮助，并且使你的应用程序看起来更专业。在这一章里，我们使用了一个简单的 `sketch` 应用程序，并且使用了工具栏、状态栏、菜单栏，通用对话框、复杂的布局、 `about` 框和启动画面来逐步对它作了改进。

2. 你可以使用一个设备上下文来直接对 `wxPython` 的显示进行绘制。不同的显示要求不同的设备上下文，然而它们共享一个通用 `API` 。为了平滑的显示，设备上下文可以被缓存。

3. 一个状态栏能够被自动地创建在框架的底部。它可以包含一个或多个文本域，各文本域可被独立调整尺寸和设置。

4. 菜单可以包含嵌套的子菜单，菜单项可以有切换状态。工具栏产生与菜单栏同种的事件，并且被设计来更易于对工具按钮分组布局。
5. 可以使用标准 `wx.FileDialog` 来管理打开和保存数据。可以使用 `wx.ColourDialog` 来选择颜色。

6. 使用 `sizer` 可以进行窗口部件的复杂布局，而不用明确地指定具体位置。 `sizer` 根据规则自动放置它的孩子对象。 `sizer` 包括的 `wx.GridSizer` 按二维网格来布局对象。 `wx.BoxSizer` 在一条线上布局项目。 `sizer` 可以嵌套，并且当 `sizer` 伸展时可以，它可以控制其孩子的行为。

7. `about` 框或别的简单的对话框可以使用 `wx.html.HtmlWindow` 来创建。启动画面用 `wx.SplashScreen` 来创建。

在第一部分（ `part` 1）中，我们已经涵盖了 `wxPython` 的基本概念，并且我们已经涉及了一些最常见的任务。在接下来的第二部分（ `part` 2）中，我们将使用目前常见的问答的格式，但是我们将涉及有关 `wxPython` 工具包的组成和功能方面的更详细的问题。

第二部分 基础wxPython

在本书的这一部分，我们将浏览基本的窗口部件，它们组成了 `wxPython` 工具包的核心。它们是你写 `wxPython` 程序要用到的关键的部分。对于其中的每个元素，我们将给你展示关于该元素的最重要的那些 `API` ，还有例子代码和关于如何在实际程序中使用该元素的技巧。

第7章，“使用基本的窗口部件”中，我们从基本窗口部件的设置开始。我们将涉及文本标签，文本域，按钮、和数字及列表选择窗口部件。我们将给你展示如何使用每个元素，如何自定义它们的外观以匹配你的应用程序，以及如何去响应用户的交互。在第8章，“在框架中放入窗口部件”中，我们将上升到容器级并谈论框架。我们将给你展示如何添加窗口部件到框架中，并说明有效的框架样式。我们也将涉及框架从创建到销毁的生命周期。在第9章，“使用对话框给用户选择”中，我们将聚焦于对话框，以对话框与框架的区别作为开始。我们也将展示一系列在 `wxPython` 中有效的预定义的对话框，以及方便使用它们的捷径。

在第十章“创建和使用 `wxPython` 菜单”中，重点是菜单。我们将讨论如何去创建菜单项，菜单项可以被附着到菜单上，可以被放置到菜单栏上。我们也将涉及切换菜单（ `toggle` `menus` ）、弹出菜单，以及各种自定义菜单显示的方法。在第11章“使用 `sizer` 放置窗口部件”中，我们将揭秘 `sizer` 技术。 `sizer` 被用来在 `wxPython` 框架和对话框中简化窗口部件的布局。我们将涉及6种预定义的 `sizer` ，给你展示它们的行为，并给出关于何时使用它们才最恰当的一些提示。最后，在第12章“处理基本的图像”中，我们将讨论经由设备上下文来在屏幕上绘图的一些基础知识，我们列出了原始的绘图方法，你可以用它们来绘制你自己的窗口部件或支持用户的绘画，或仅用于装饰。

第七章 使用基础控件

`wxPython` 工具包提供了多种不同的窗口部件，包括了本章所提到的基本控件。我们涉及静态文本、可编辑的文本、按钮、微调、滑块、复选框、单选按钮、选择器、列表框、组合框和标尺。对于每种窗口部件，我们将提供一个关于如何使用它的简短例子，并附上相关的 `wxPython` `API` 的说明。

显示文本

这一节以在屏幕上显示文本的例子作为开始，包括用作标签的静态文本域，有样式和无样式的都使用了。你可以创建用于用户输入的单行和多行文本域。另外，我们将讨论如何选择文本的字体。

如何显示静态文本？

大概对于所有的 `UI` 工具来说，最基本的任务就是在屏幕上绘制纯文本。
在 `wxPython` 中，使用类 `wx.StaticText` 来完成。图7.1显示了这个静态文本控件。

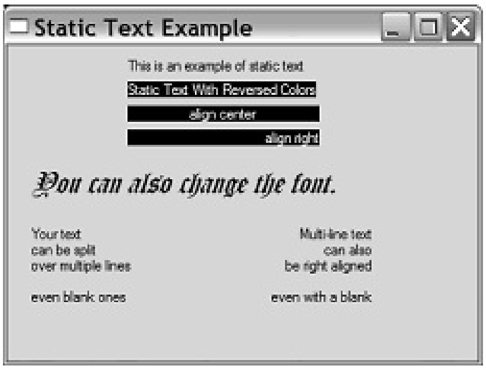


Figure 7.1 Samples of `wx.StaticText`, including font, alignment, and color changes

在 `wx.StaticText` 中，你能够改变文本的对齐方式、字体和颜色。简单的静态文本控件可以包含多行文本，但是你不能处理多种字体或样式。处理多种字体或样式，要使用更精细的文本控件，如 `wx.html.HTMLWindow` ，它在第十六章中说明。为了在静态文本控件中显示多行文本，我们要包括其中有换行符的字符串，并使控件的大小足够显示所有的文本。有一个特点是你在图7.1中所不能看到的，那就是 `wx.StaticText` 窗口不会接受或响应鼠标事件。

如何显示静态文本

例子7.1显示了产生图7.1的代码。

例**7.1** 如何使用静态文本的一个基本例子

```
import wx

class StaticTextFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Static Text Example',
                size=(400, 300))
        panel = wx.Panel(self, -1)

# 这是一个基本的静态文本
wx.StaticText(panel, -1, "This is an example of static text",
                (100, 10))

# 指定了前景色和背景色的静态文本
rev = wx.StaticText(panel, -1, "Static Text With Reversed Colors
",
                (100, 30))
        rev.SetForegroundColour('white')
        rev.SetBackgroundColour('black')

# 指定居中对齐的的静态文本
center = wx.StaticText(panel, -1, "align center", (100, 50),
                (160, -1), wx.ALIGN_CENTER)
        center.SetForegroundColour('white')
        center.SetBackgroundColour('black')

# 指定右对齐的静态文本
right = wx.StaticText(panel, -1, "align right", (100, 70),
                (160, -1), wx.ALIGN_RIGHT)
        right.SetForegroundColour('white')
        right.SetBackgroundColour('black')

# 指定新字体的静态文本
        str = "You can also change the font."
        text = wx.StaticText(panel, -1, str, (20, 100))
        font = wx.Font(18, wx.DECORATIVE, wx.ITALIC, wx.NORMAL)
        text.SetFont(font)

# 显示多行文本
        wx.StaticText(panel, -1, "Your text\ncan be split\n"
                "over multiple lines\n\neven blank ones", (20,15
0))

#显示对齐的多行文本
        wx.StaticText(panel, -1, "Multi-line text\ncan also\n"
"be right aligned\n\neven with a blank", (220,150),
                style=wx.ALIGN_RIGHT)
```

```
if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = StaticTextFrame()
    frame.Show()
    app.MainLoop()
```

`wx.StaticText` 的构造函数和基本的 `wxWidget` 构造函数相同，如下所示：

```
wx.StaticText(parent, id, label, pos=wx.DefaultPosition,
size=wx.DefaultSize, style=0, name="staticText")
```

表7.1说明了这些参数——大多数的 `wxPython` 窗口部件都有相类似的参数。对于构造函数的参数的更详细的说明，请参见第2章的相关论述。

表**7.1** **`wx.StaticText`** 构造函数的参数

`parent` ：父窗口部件。

`id` ：标识符。使用-1可以自动创建一个唯一的标识。

`label` ：你想显示在静态控件中的文本。

`pos` ：一个 `wx.Point` 或一个 `Python` 元组，它是窗口部件的位置。

`size` ：一个 `wx.Size` 或一个 `Python` 元组，它是窗口部件的尺寸。

`style` ：样式标记。

`name` ：对象的名字，用于查找的需要。

接下来我们更详细地讨论样式标记。

使用样式工作

所有在例7.1中静态文本实例所调用的方法都是属于基父类 `wx.Window` 的； `wx.StaticText` 没有定义任何它自己的新方法。表7.2列出了一些专用于 `wx.StaticText` 的样式。

表**7.2**

`wx.ALIGN_CENTER` ：静态文本位于静态文本控件的中心。

`wx.ALIGN_LEFT` ：文本在窗口部件中左对齐。这是默认的样式。

`wx.ALIGN_RIGHT` ：文本在窗口部件中右对齐。

`wx.ST_NO_AUTORESIZE` ：如果使用了这个样式，那么在使用了 `SetLabel()` 改变文本之后，静态文本控件不将自我调整尺寸。你应结合使用一个居中或右对齐的控件来保持对齐。

wx.StaticText 控件覆盖了 SetLabel() ，以便根据新的文本来调整自身，除非 wx.ST_NO_AUTORESIZE 样式被设置了。

当创建了一个居中或右对齐的单行静态文本时，你应该显式地在构造器中设置控件的尺寸。指定尺寸以防止 wxPython 自动调整该控件的尺寸。 wxPython 的默认尺寸是刚好包容了文本的矩形尺寸，因此对齐就没有什么必要。要在程序中动态地改变窗口部件中的文本，而不改变该窗口部件的尺寸，就要设置 wx.ST_NO_AUTORESIZE 样式。这样就防止了在文本被重置后，窗口部件自动调整尺寸到刚好包容了文本。如果静态文本是位于一个动态的布局中，那么改变它的尺寸可能导致屏幕上其它的窗口部件移动，这就对用户产生了干扰。

其它显示文本的技术

还有其它的方法来显示文本。其中之一就是 wx.lib.stattext.GenStaticText 类，它是 wx.StaticText 的纯 Python 实现。它比标准C++版的跨平台性更好，并且它接受鼠标事件。当你想子类化或创建你自己的静态文本控件时，它是更可取的。

你可以使用 DrawText(text , x, y) 和 DrawRotatedText(text , x, y, angle) 方法直接绘制文本到你的设备上下文。后者是显示有一定角度的文本的最容易的方法，尽管 GenStaticText 的子类也能处理旋转问题。设备上下文在第6章中做了简短的说明，我们将在第12章中对它做更详细的说明。

如何让用户输入文本？

超越纯粹显示静态文本，我们将开始讨论当输入文本时的用户交互。 wxPython 的文本域窗口部件的类是 wx.TextCtrl ，它允许单行和多行文本输入。它也可以作为密码输入控件，掩饰所按下的按键。如果平台支持的话， wx.TextCtrl 也提供丰富格式文本的显示，通过使用所定义和显示的多文本样式。图7.2显示了一个作为单行控件的 wx.TextCtrl 的样板。其中的密码输入框对密码进行了掩饰。



Figure 7.2 Examples of the single line text control, both plain and password

接下来，我们将演示如何创建文本，然后讨论文本控件的样式选项。

如何创建文本输入控件

例子7.2显示了用于生成图7.2的代码

例**7.2** **wx.TextCtrl** 的单行例子

```
import wx

class TextFrame(wx.Frame):

    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Text Entry Example',
                size=(300, 100))
panel = wx.Panel(self, -1)
        basicLabel = wx.StaticText(panel, -1, "Basic Control:")
basicText = wx.TextCtrl(panel, -1, "I've entered some text!",
                size=(175, -1))
        basicText.SetInsertionPoint(0)

        pwdLabel = wx.StaticText(panel, -1, "Password:")
pwdText = wx.TextCtrl(panel, -1, "password", size=(175, -1),
                style=wx.TE_PASSWORD)
        sizer = wx.FlexGridSizer(cols=2, hgap=6, vgap=6)
        sizer.AddMany([basicLabel, basicText, pwdLabel, pwdText]
)
        panel.SetSizer(sizer)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
    app.MainLoop()
```

`wx.TextCtrl` 类的构造函数较小且比其父类 `wx.Window` 更精细，它增加了两个参数：

`wx.TextCtrl(parent` , `id` , `value` = "", `pos` = `wx.DefaultPosition` , `size` = `wx.DefaultSize` , `style` =0, `validator` = `wx.DefaultValidator` `name` = `wx.TextCtrlNameStr)`

参数 `parent` , `id` , `pos` , `size` , `style` , 和 `name` 与 `wx.Window` 构造函数的相同。 `value` 是显示在该控件中的初始文本。

`validator` 参数用于一个 `wx.Validator` 。 `validator` 通常用于过虑数据以确保只能键入要接受的数据。将在第9章对 `validator` 做更详细的讨论。

使用单行文本控件样式

这里，我们将讨论一些唯一无二的文本控件样式。 表7.3说明了用于单行文本控件的样式标记

表7.3 单行 **`wx.TextCtrl`** 的样式

`wx.TE_CENTER` ：控件中的文本居中。

`wx.TE_LEFT` ：控件中的文本左对齐。默认行为。

`wx.TE_NOHIDESEL` ：文本始终高亮显示，只适用于`Windows`。

`wx.TE_PASSWORD` ：不显示所键入的文本，代替以星号显示。

`wx.TE_PROCESS_ENTER` ：如果使用了这个样式，那么当用户在控件内按下回车键时，一个文本输入事件被触发。否则，按键事件内在的由该文本控件或该对话框管理。

`wx.TE_PROCESS_TAB` ：如果指定了这个样式，那么通常的字符事件在`Tab`键按下时创建（一般意味一个制表符将被插入文本）。否则，`tab`由对话框来管理，通常是控件间的切换。

`wx.TE_READONLY` ：文本控件为只读，用户不能修改其中的文本。

`wx.TE_RIGHT` ：控件中的文本右对齐。

像其它样式标记一样，它们可以使用|符号来组合使用，尽管其中的三个对齐标记是相互排斥的。

对于添加文本和移动插入点，该文本控件自动管理用户的按键和鼠标事件。对于该文本控件可用的命令控制组合说明如下：

- `ctrl`-x ：剪切 `ctrl`-c ：复制 `ctrl`-v ：粘贴 `ctrl`-z ：撤消

不输入的情况下如何改变文本？

除了根据用户的输入改变显示的文本外，`wx.TextCtrl`提供了在程序中改变显示的文本的一些方法。你可以完全改变文本或仅移动插入点到文本中不同的位置。表7.4列出了`wx.TextCtrl`的文本处理方法。

表**7.4**

`AppendText(text)` ：在尾部添加文本。

`Clear()` ：重置控件中的文本为“”。并且生成一个文本更新事件。

`EmulateKeyPress(event)` ：产生一个按键事件，插入与事件相关联的控制符，就如同实际的按键发生了。

`GetInsertionPoint()` `SetInsertionPoint(pos)` `SetInsertionPointEnd()` ：得到或设置插入点的位置，位置是整型的索引值。控件的开始位置是0。

`GetRange(from`, `to)` ：返回控件中位置索引范围内的字符串。

`GetSelection()` `GetStringSelection()` `SetSelection(from`, `to)` ：`GetSelection()`以元组的形式返回当前所选择的文本的起始位置的索引值（开始，结束）。`GetStringSelection()`得到所选择的字符串。`SetSelection(from`, `to)`设置选择的文本。

`GetValue()` `SetValue(value)` ：`SetValue()` 改变控件中的全部文本。`GetValue()` 返回控件中所有的字符串。

`Remove(from` , `to)` ：删除指定范围的文本。

`Replace(from` , `to` , `value)` ：用给定的值替换掉指定范围内的文本。这可以改变文本的长度。

`WriteText(text)` ：类似于 `AppendText()` ，只是写入的文本被放置在当前的插入点。

当你的控件是只读的或如果你根据事件而非用户键盘输入来改变控件中的文本是，这些方法是十分有用的。

如何创建一个多行或样式文本控件?

你可以使用 `wx.TE_MULTILINE` 样式标记创建一个多行文本控件。如果本地窗口控件支持样式，那么你可以改变被控件管理的文本的字体和颜色样式，这有时被称为丰富格式文本。对于另外的一些平台，设置样式的调用被忽视掉了。图7.3显示了多行文本控件的一个例子。

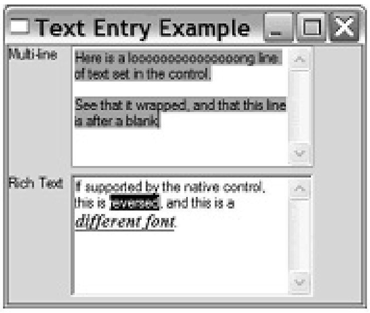


Figure 7.3 Examples of multi-line text controls, both with and without rich text

例7.3包含了用于创建图7.3的代码。通常，创建一个多行文本控件是通过设置 `wx.TE_MULTILINE` 样式标记来处理的。较后的部分，我们将讨论使用丰富文本样式。

例**7.3** 创建一个多行文本控件

```
import wx

class TextFrame(wx.Frame):

    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Text Entry Example',
                size=(300, 250))
panel = wx.Panel(self, -1)
        multiLabel = wx.StaticText(panel, -1, "Multi-line")
        multiText = wx.TextCtrl(panel, -1,
               "Here is a looooooooooooooong line of text set in
 the control.\n\n"
               "See that it wrapped, and that this line is after
 a blank",
               size=(200, 100), style=wx.TE_MULTILINE) #创建一个文
本控件
        multiText.SetInsertionPoint(0) #设置插入点

        richLabel = wx.StaticText(panel, -1, "Rich Text")
richText = wx.TextCtrl(panel, -1,
                "If supported by the native control, this is rev
ersed, and this is a different font.",
                size=(200, 100), style=wx.TE_MULTILINE|wx.TE_RIC
H2) #创建丰富文本控件
        richText.SetInsertionPoint(0)
        richText.SetStyle(44, 52, wx.TextAttr("white", "black"))
 #设置文本样式
points = richText.GetFont().GetPointSize()
        f = wx.Font(points + 3, wx.ROMAN, wx.ITALIC, wx.BOLD, Tr
ue) #创建一个字体
        richText.SetStyle(68, 82, wx.TextAttr("blue", wx.NullCol
our, f)) #用新字体设置样式
        sizer = wx.FlexGridSizer(cols=2, hgap=6, vgap=6)
        sizer.AddMany([multiLabel, multiText, richLabel, richTex
t])
        panel.SetSizer(sizer)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
    app.MainLoop()
```

使用多行或丰富文本样式

除了 `wx.TE_MULTILINE` ，还有另外的样式标记，它们只在一个多行或丰富文本控件的上下文中有意义。表7.5列出了这些窗口样式。

表7.5

`wx.HSCROLL`：如果文本控件是多行的，并且如果该样式被声明了，那么长的行将不会自动换行，并显示水平滚动条。该选项在`GTK`+中被忽略。

`wx.TE_AUTO_URL`：如果丰富文本选项被设置并且平台支持的话，那么当用户的鼠标位于文本中的一个`URL`上或在该`URL`上敲击时，这个样式将导致一个事件被生成。

`wx.TE_DONTWRAP`：`wx.HSCROLL`的别名。

`wx.TE_LINEWRAP`：对于太长的行，以字符为界换行。某些操作系统可能会忽略该样式。

`wx.TE_MULTILINE`：文本控件将显示多行。

`wx.TE_RICH`：用于`Windows`下，丰富文本控件用作基本的窗口部件。这允许样式文本的使用。

`wx.TE_RICH2`：用于`Windows`下，把最新版本的丰富文本控件用作基本的窗口部件。

`wx.TE_WORDWRAP`：对于太长的行，以单词为界换行。许多操作系统会忽略该样式。

记住，上面这些样式可以组合使用，所以上面例子中的多行丰富文本控件使用`wx.TE_MULTILINE` | `wx.TE_RICH2`来声明。

用在`wx.TextCtrl`窗口部件中的文本样式是类`wx.TextAttr`的实例。`wx.TextAttr`实例的属性有文本颜色、背景色、和字体，它们都能够在构造函数中被指定，如下所示：

`wx.TextAttr(colText`, `colBack` = `wx.NullColor`, `font` = `wx.NullFont)`

文本色和背景色是`wxPython`对象，它们可以使用颜色名或颜色的`RGB`值(红, 绿, 蓝)来指定。`wx.NullColor`指明使用控件目前的背景色。`font`是一个`wx.Font`对象，我们将在下一小节讨论。`wx.NullFont`对象指明使用当前默认字体。

类`wx.TextAttr`有相关属性的`get`*()*方法：*`GetBackgroundColour()`*, *`GetFont()`*, 和 *`GetTextColour()`*，也有返回布尔值的验证存在性的方法：*`HasBackgroundColour()`*, *`HasFont()`*, 和 *`HasTextColour()`*。如果属性包含一个默认值，则*`Has`*()方法返回`False`。如果所有这三个属性都包含默认值，则`IsDefault()`方法返回`true`。这个类没有`set`*()方法，因为`wx.TextAttr`的实例是不可变的。要改变文本的样式，你必须创建一个实例。

使用文本样式，要调用`SetDefaultStyle(style)`或`SetStyle(start`, `end`, `style)`。第一个方法设置为控件当前的样式。任何插入到该控件中的文本，不管是键入的，或使用了`AppendText()`或`WriteText()`方法的，都以该样式显示。如果样式的某个属性是默认的，那么该样式的当前值被保留。但是，如果样式的所有属性都是默认的，那么恢复默认样

式。 `SetStyle()` 与 `SetDefaultStyle(style)` 类似，只是立即对位于 `start` 和 `end` 位置之间的文本起作用。样式参数中的默认属性通过检查该控件的当前默认样式来解决。例7.3使用下面一行代码来反转文本中几个字符的颜色：

`richText.SetStyle(44`, 52, `wx.TextAttr(` " `white` ", " `black` "))

背景色变为了黑色，相应的字符变为了白色。

表7.6列出了 `wx.TextCtrl` 的方法，它们在处理多行控件和丰富文本中是有用的。

表**7.6**

`GetDefaultStyle()` `SetDefaultStyle(style)` ：上面已作了说明。

`GetLineLength(lineNo)` ：返回给定行的长度的整数值。

`GetLineText(lineNo)` ：返回给定行的文本。

`GetNumberOfLines()` ：返回控件中的行的数量。对于单行，返回1。

`IsMultiLine()` `IsSingleLine()` ：布尔类型的方法，确定控件的状态。

`PositionToXY(pos)` ：指定文本内的一个整数值位置，返回以元组(列，行)形式的索引位置。列和行的索引值均以0作为开始。

`SetStyle(start`, `end`, `style)` ：立即改变指定范围内文本的样式。

`ShowPosition(pos)` ：引起一个多行控件的滚动，以便观察到指定位置的内容。

`XYToPosition(x`, `y)` ：与 `PositionToXY(pos)` 相反——指定行和列，返回整数值位置。

如果你能在系统中使用任意字体的话，那么就可以更加灵活的创建样式。 接下来，我们将给你展示如何创建和使用字体实例。

如何创建一个字体？

字体是类 `wx.Font` 的实例。你所访问的任何字体，它已经被安装并对于基本的系统是可访问的。创建一个字体实例，要使用如下的构造函数：

`wx.Font(pointSize`, `family`, `style`, `weight`, `underline` = `False`, `faceName` ="", `encoding` = `wx.FONTENCODING_DEFAULT)`

`pointSize` 是字体的以磅为单位的整数尺寸。 `family` 用于快速指定一个字体而无需知道该字体的实际的名字。字体的准确选择依赖于系统和具体可用的字体。可用的字体类别的示例显示在表7.7中。你所得到的精确的字体将依赖于你的系统。

表**7.7**

`wx.DECORATIVE` ：一个正式的，老的英文样式字体。

`wx.DEFAULT` ：系统默认字体。

`wx.MODERN`：一个单间隔（固定字符间距）字体。

`wx.ROMAN`：`serif` 字体，通常类似于 `Times` `New` `Roman`。

`wx.SCRIPT`：手写体或草写体

`wx.SWISS`：`sans` - `serif` 字体，通常类似于 `Helvetica` 或 `Arial`。

`style` 参数指明字体的是否倾斜，它的值有：`wx.NORMAL`，`wx.SLANT`，和 `wx.ITALIC`。同样，`weight` 参数指明字体的醒目程度，可选值有：`wx.NORMAL`，`wx.LIGHT`，或 `wx.BOLD`。这些常量值的行为根据它的名字就可以知道了。`underline` 参数仅工作在 `Windows` 系统下，如果取值为 `True`，则加下划线，`False` 为无下划线。`faceName` 参数指定字体名。

`encoding` 参数允许你在几个编码中选择一个，它映射内部的字符和字本显示字符。编码不是 `Unicode` 编码，只是用于 `wxPython` 的不同的8位编码。大多数情况你可以使用默认编码。

为了获取系统的有效字体的一个列表，并使用户可用它们，要使用专门的类 `wx.FontEnumerator`，如下所示：

e = `wx.FontEnumerator()` `e.EnumerateFacenames()` `fontList` = `e.GetFacenames()`

要限制该列表为固定宽度，就要将上面的第一行改为e = `wx.FontEnumerator(fixedWidth` = `True)`。

如果我们系统不支持丰富文本，那么我还能使用样式文本吗？

可以。在 `wxPython` 中有一个跨平台的样式文本窗口部件，名为 `wx.stc.StyledTextCtrl`，它是 `Python` 对 `Scintilla` 丰富文本组件的封装。因为 `Scintilla` 不是 `wxWidgets` 的一部分，而是作为一个独立的第三方组被合并到了 `wxPython` 中，所以它不与我们已经讨论过的类共享相同的 `API`。`wx.stc.StyledCtrl` 的完整说明超过了我们要讲的范围，但是你可以在 `http:` // `wiki.wxpython.org` / `index.cgi` / `wxStyledTextCtrl` 找到相关的文档。

如果我的文本控件不匹配我的字符串该怎么办？

当使用多行 `wx.TextCtrl` 的时候，要知道的一点是，该文本控件是以何种方式存储字符串的。在内部，存储在该 `wx.TextCtrl` 中的多行字符是以\n作为行的分隔符的。这与基本的操作系统无关，即使某些系统使用了不同的字符组合作为一行的分隔符。当你使用 `GetValue()` 来获取该字符串时，原来的行分隔符被还原，因此你不必考虑手工转换。这个的好处就是控件中的文本不依赖于任何特定的操作系统。

缺点是，文本控件中的行的长度和行的索引与它们在文本控件外的可能是不同的。例如，如果你在一个 `Windows` 系统上，系统所用的行分隔符是\r\n，通过 `GetValue()` 所得知的字符串的长度将比通过 `GetLastPosition()` 所得知的

字符串的结尾长。通过在例7.3中增加下面两行：

`print` " `getValue` ", `len(multiText.GetValue())` `print` " `lastPos` ", `multiText.GetLastPosition()`

我们在 `Unix` 系统上所得的结果应该是：

`getValue` 119 `lastPos` 119

我们在 `Windows` 系统上所得的结果应该是：

`getValue` 121 `lastPos` 119

这意味你不应该使用多行文本控件的位置索引来取得原字符串，位置索引应该用作 `wx.TextCtrl` 的另外方法的参数。对于该控件中的文本的子串，应该使用 `GetRange()` 或 `GetSelectedText()` 。也不要反向索引；不要使用原字符串的索引来取得并放入文本控件中。下面是一个例子，它使用了不正确的方法在插入点之后直接得到10个字符：

```
aLongString = """Any old
multi line string
will do here.
Just as long as
it is multiline"""
text = wx.TextCtrl(panel, -1, aLongString, style=wx.TE_MULTILINE
)
x = text.GetInsertionPoint()
selection = aLongString[x : x + 10] ### 这将是不正确的
```

在 `Windows` 或 `Mac` 系统中要得到正确的结果，最后一行应换为：

`selection` = `text.GetRange(x` , x + 10)

如何响应文本事件？

有一个由 `wx.TextCtrl` 窗口部件产生的便利的命令事件，你可能想用它。你需要把相关事件传递给 `Bind` 方法以捕获该事件，如下所示：

`frame.Bind(wx.EVT_TEXT` , `frame.OnText` , `text)`

表7.8说明了这些命令事件。

表7.8 `wx.TextCtrl` 的事件

`EVT_TEXT` ：当控件中的文本改变时产生该事件。文本因用户的输入或在程序中使用 `SetValue()` 而被改变，都要产生该事件。

`EVT_TEXT_ENTER` ：当用户在一个 `wx.TE_PROCESS_ENTER` 样式的文本控件中按下了回车键时，产生该事件。

`EVT_TEXT_URL`：如果在`Windows`系统上，`wx.TE_RICH`或`wx.TE_RICH2`样式被设置了，并且`wx.TE_AUTO_URL`样式也被设置了，那么当在文本控件内的`URL`上发生了一个鼠标事件时，该事件被触发。

`EVT_TEXT_MAXLEN`：如果使用`SetMaxLength()`指定了该控件的最大长度，那么当用户试图输入更长的字符串时，该事件被触发。你可能会用这个，例如，这时给用户显示一个警告消息。

接下来，让我们来讨论被主要设计来得到鼠标输入的控件。其中最简单的就是按钮。

使用按钮工作

在`wxPython`中有很多不同类型的按钮。这一节，我们将讨论文本按钮、位图按钮、开关按钮（`toggle` `buttons`）和通用（`generic`）按钮。

如何生成一个按钮？

在第一部分（`part` 1）中，我们已经说明了几个按钮的例子，所以这里我们只简短的涉及它的一些基本的东西。图7.4显示了一个简单的按钮。

图**7.4**



Figure 7.4 A simple button

使用按钮是非常简单的。例7.4显示了该简单按钮的代码。

例**7.4** 创建并显示一个简单的按钮

```
import wx

class ButtonFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Button Example',
                size=(300, 100))
        panel = wx.Panel(self, -1)
        self.button = wx.Button(panel, -1, "Hello", pos=(50, 20)
)
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button)
        self.button.SetDefault()

    def OnClick(self, event):
        self.button.SetLabel("Clicked")

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ButtonFrame()
    frame.Show()
    app.MainLoop()
```

`wx.Button` 的构造函数类似于我们已经看到过的，如下所示：

`wx.Button(parent` , `id` , `label` , `pos` , `size` = `wxDefaultSize` , `style` =0, `validator` , `name` =" `button` ")

参数 `label` 是显示在按钮上的文本。它可以在程序运行期间使用 `SetLabel()` 来改变，并且使用 `GetLabel()` 来获取。另外两个有用的方法是 `GetDefaultSize()` 和 `SetDefault()` 。 `GetDefaultSize()` 返回系统默认按钮的尺寸（对于框架间的一致性是有用的）； `SetDefault()` 设置按钮为对话框或框架的默认按钮。默认按钮的绘制不同于其它按钮，它在对话框获得焦点时，通常按下回车键被激活。

`wx.Button` 类有一个跨平台的样式标记： `wx.BU_EXACTFIT` 。如果定义了这个标记，那么按钮就不把系统默认的尺寸作为最小的尺寸，而是把能够恰好填充标签的尺寸作为最小尺寸。如果本地窗口部件支持的话，你可以使用标记 `wx.BU_LEFT` , `wx.BU_RIGHT` , `wx.BU_TOP` , 和 `wx.BU_BOTTOM` 来改变按钮中标签的对齐方式。每个标记对齐标签到边，该边你根据标记的名字可以知道。正如我们在第一部分中所讨论过的， `wx.Button` 在被敲击时触发一个命令事件，事件类型是 `EVT_BUTTON` 。

如何生成一个位图按钮？

有时候，你可能想在你的按钮上显示一个图片，而非一个文本标签，如图7.5所示。



Figure 7.5 A demonstration of a basic bitmap button. The left button is drawn with a 3D effect.

在 `wxPython` 中，使用类 `wx.BitmapButton` 来创建一个位图按钮。处理一个 `wx.BitmapButton` 的代码是与通用按钮的代码非常类似的，例7.5显示了产生7.5的代码。

例7.5 创建一个位图按钮

```
import wx

class BitmapButtonFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Bitmap Button Example',
                size=(200, 150))
        panel = wx.Panel(self, -1)
        bmp = wx.Image("bitmap.bmp", wx.BITMAP_TYPE_BMP).Convert
ToBitmap()
        self.button = wx.BitmapButton(panel, -1, bmp, pos=(10, 2
0))
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button)
        self.button.SetDefault()
self.button2 = wx.BitmapButton(panel, -1, bmp, pos=(100, 20),
            style=0)
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button2)

    def OnClick(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = BitmapButtonFrame()
    frame.Show()
    app.MainLoop()
```

与普通按钮的主要的区别是你需要提供一个位图，而非一个标签。否则，构造器和大部分代码是与文本按钮的例子相同的。位图按钮在被敲击时同样产生 `EVT_BUTTON` 事件。

关于位图按钮有几个有趣的特性。首先，一个样式标记 `wx.BU_AUTODRAW` ，它是默认的。如果该标记是打开的，那么位图将带有一个3D的边框，这使它看起来像一个文本按钮（图7.5中的左按钮），并且按钮比原位图大几个像素。如果该标记是关闭的，则位图被简单地绘制为按钮而没有边框。通过设置 `style` =0使图7.5中右边的按钮关闭默认设置，它没有了3D的效果。

默认情况下，给 `wxPython` 传递单个位图作为主显示的位图，在当按钮被按下或获得焦点或无效时， `wxPython` 自动创建一个标准的派生自主显示的位图的位图作为此时显示在按钮上的位图。如果自动创建的位图不是你想要的，你可以使用下面的方法： `SetBitmapDisabled()` , `SetBitmapFocus()` , `SetBitmapLabel()` , 和 `SetBitmap` - `Selected()` 显式地告诉 `wxPython` 你要使用哪个位图。这些方法都要求一个 `wx.Bitmap` 对象作为参数，并且它们都有相应的 `get` *()方法。

你不能通过使用标准的 `wxWidgets` C++库来合并一个位图和文本。你可以创建一个包含文本的位图。然而，正如我们将在通用按钮问题讨论中所看到的， `wxPython` 有额外的方法来实现这一合并行为。

=== 如何创建开关按钮（ `toggle` `button` ）？===

你可以使用 `wx.ToggleButton` 创建一个开关按钮（ `toggle` `button` ）。开关按钮（ `toggle` `button` ）看起来十分像文本按钮，但它的行为更像复选框，它的选择或非选择状态是可视化的。换句话说，当你按下一个开关按钮（ `toggle` `button` ）时，它将一直保持被按下的状态直到你再次敲击它。

在 `wx.ToggleButton` 与父类 `wx.Button` 之间只有丙个区别：

1、当被敲击时， `wx.ToggleButton` 发送一个 `EVT_TOGGLEBUTTON` 事件。
2、 `wx.ToggleButton` 有 `GetValue()` 和 `SetValue()` 方法，它们处理按钮的二进制状态。

开关按钮（ `toggle` `button` ）是有用的，它相对于复选框是另一好的选择，特别是在工具栏中。记住，你不能使用 `wxWidgets` 提供的对象来将开关按钮（ `toggle` `button` ）与位图按钮合并，但是 `wxPython` 有一个通用按钮类，它提供了这种行为，我们将在下一节对其作讨论。

什么是通用按钮，我为什么要使用它？

通用按钮是一个完全用 `Python` 重新实现的一个按钮窗口部件，回避了本地系统窗口部件的用法。它的父类是 `wx.lib.buttons.` `GenButton` 。通用按钮有通用位图和切换按钮。

这儿有几个使用通用按钮的原因：

1、通用按钮比本地按钮具有更好的跨平台的外观。另一方面，通用按钮可能在具体的系统上看起来与本地按钮有些微的不同。

2、使用通用按钮，你对它的外观有更多的控制权，并且能改变属性，如3D斜面的宽度和颜色，而这对于本地控件可能是不允许的。

3、通用按钮类允许特性的合并，而 `wxWidget` 按钮不行。比如 `GenBitmapTextButton` 允许文本标签和位图的组合， `GenBitmapToggleButton` 实现一个位图切换按钮。

4、如果你正在创建一个按钮类，使用通用按钮是较容易的。由于其代码和参数是用 `Python` 写的，所以当创建一个新的子类的时候，对于检查和覆盖，它们的可用性更好。

图7.6显示了实际的通用按钮和常规按钮的对照。

图**7.6**

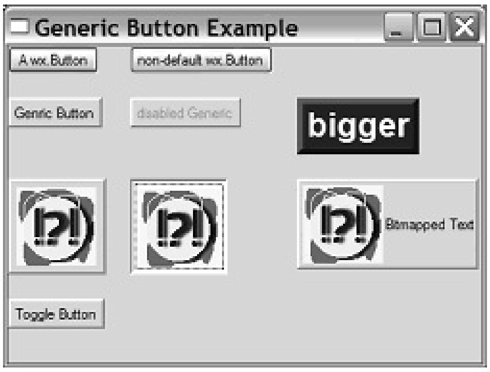


Figure 7.6
Generic buttons. The top row has regular buttons for contrast. This shows different color combinations, bitmap, bitmap toggle, and bitmap text buttons.

例7.6显示了产生图7.6的代码。第二个导入语句：`import wx.lib.buttons as buttons`，是必须的，它使得通用按钮类可用。

例**7.6** 创建和使用 `wxPython` 的通用按钮

```
import wx
import wx.lib.buttons as buttons

class GenericButtonFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Generic Button Example',
                size=(500, 350))
        panel = wx.Panel(self, -1)

        sizer = wx.FlexGridSizer(1, 3, 20, 20)
        b = wx.Button(panel, -1, "A wx.Button")
        b.SetDefault()
        sizer.Add(b)

        b = wx.Button(panel, -1, "non-default wx.Button")
        sizer.Add(b)
        sizer.Add((10,10))

        b = buttons.GenButton(panel, -1, 'Genric Button')#基本的
通用按钮
        sizer.Add(b)

        b = buttons.GenButton(panel, -1, 'disabled Generic')#无效
的通用按钮
        b.Enable(False)
        sizer.Add(b)

        b = buttons.GenButton(panel, -1, 'bigger')#自定义尺寸和颜色
的按钮
        b.SetFont(wx.Font(20, wx.SWISS, wx.NORMAL, wx.BOLD, Fals
e))
        b.SetBezelWidth(5)
```

```
        b.SetBackgroundColour("Navy")
        b.SetForegroundColour("white")
        b.SetToolTipString("This is a BIG button...")
sizer.Add(b)

        bmp = wx.Image("bitmap.bmp", wx.BITMAP_TYPE_BMP).Convert
ToBitmap()
        b = buttons.GenBitmapButton(panel, -1, bmp)#通用位图按钮
        sizer.Add(b)

        b = buttons.GenBitmapToggleButton(panel, -1, bmp)#通用位
图开关按钮
        sizer.Add(b)

        b = buttons.GenBitmapTextButton(panel, -1, bmp, "Bitmapp
ed Text",
                size=(175, 75))#位图文本按钮
        b.SetUseFocusIndicator(False)
        sizer.Add(b)

        b = buttons.GenToggleButton(panel, -1, "Toggle Button")#
通用开关按钮
        sizer.Add(b)

        panel.SetSizer(sizer)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = GenericButtonFrame()
    frame.Show()
app.MainLoop()
```

在例7.6中，通用按钮的用法非常类似于常规按钮。通用按钮产生与常规按钮同样的 `EVT_BUTTON` 和 `EVT_TOGGLEBUTTON` 事件。通用按钮引入了 `GetBevelWidth()` 和 `SetBevelWidth()` 方法来改变3D斜面效果。它们用在了图7.6中大按钮上。

通用位图按钮类 `GenBitmapButton` 工作的像标准的 `wxPython` 版本。在构造器中。 `GenBitmapTextButton` 要求先要一个位图，然后是文本。通用类 `GenToggleButton` , `GenBitmapToggleButton` ,和 `GenBitmapTextToggleButton` 与非开关版的一样，并且对于处理按钮的开关状态响应于 `GetToggle()` 和 `SetToggle()` 。

在下一节，我们将讨论关于使你的用户能够输入或观看一个数字值的方案。

输入并显示数字

有时你想要显示图形化的数字信息，或你想让用户不必使用键盘来输入一个数字量。在这一节，我们将浏览 `wxPython` 中用于数字输入和显示的工具：滑块（ `slider` ）、微调控制框和显示量度的标尺。

如何生成一个滑块？

滑块是一个窗口部件，它允许用户通过在该控件的尺度内拖动指示器来选择一个数值。在 `wxPython` 中，该控件类是 `wx.Slider` ，它包括了滑块的当前值的只读文本的显示。图7.7显示了水平和垂直滑块的例子。

图**7.7**

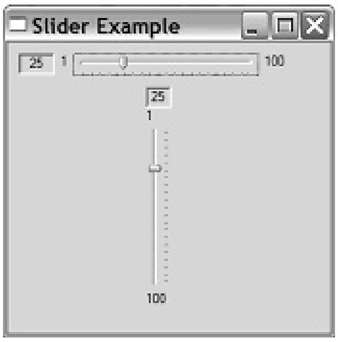


Figure 7.7 A vertical wx.Slider and a horizontal wx.Slider, which use the style flag wx.SL_LABELS

滑块的基本使用是十分简单的，但是你可以增加许多事件。

如何使用滑块

例7.7是产生图7.7的例子。

例**7.7** 水平和垂直滑块的显示代码

```
import wx

class SliderFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Slider Example',
                size=(300, 350))
        panel = wx.Panel(self, -1)
        self.count = 0
        slider = wx.Slider(panel, 100, 25, 1, 100, pos=(10, 10),
                size=(250, -1),
                style=wx.SL_HORIZONTAL | wx.SL_AUTOTICKS | wx.SL
_LABELS )
        slider.SetTickFreq(5, 1)
        slider = wx.Slider(panel, 100, 25, 1, 100, pos=(125, 70)
,
                size=(-1, 250),
                style=wx.SL_VERTICAL | wx.SL_AUTOTICKS | wx.SL_L
ABELS )
        slider.SetTickFreq(20, 1)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SliderFrame()
    frame.Show()
app.MainLoop()
```

通常，当你使用 `wx.Slider` 类时，所有你所需要的就是一个构造函数，它与别的调用不同，如下所示：

`wx.Slider(parent` , `id` , `value` , `minValue` , `maxValue` , `pos` = `wxDefaultPosition` , `size` = `wx.DefaultSize` , `style` = `wx.SL_HORIZONTAL` , `validator` = `wx.DefaultValidator` , `name` =" `slider` ")

`value` 是滑块的初始值，而 `minValue` 和 `maxValue` 是两端的值。

使用滑块样式工作

滑块的样式管理滑块的位置和方向，如下表7.9所示。

表7.9 `wx.Slider` 的样式

`wx.SL_AUTOTICKS` ：如果设置这个样式，则滑块将显示刻度。刻度间的间隔通过 `SetTickFreq` 方法来控制。

`wx.SL_HORIZONTAL` ：水平滑块。这是默认值。

`wx.SL_LABELS` ：如果设置这个样式，那么滑块将显示两头的值和滑块的当前只读值。有些平台可能不会显示当前值。

`wx.SL_LEFT` ：用于垂直滑块，刻度位于滑块的左边。

`wx.SL_RIGHT`：用于垂直滑块，刻度位于滑块的右边。

`wx.SL_TOP`：用于水平滑块，刻度位于滑块的上部。

`wx.SL_VERTICAL`：垂直滑块。

如果你想通过改变滑块中的值来影响你的应用程序中的其它的部分，那么这儿有几个你可使用的事件。这些事件与窗口滚动条所发出的是相同的，详细的说明参见第8章的滚动条部分。

表7.10列出了你可用于滑块的`Set`*()*方法。每个`Set`()方法都有一个对应的`Get`方法——`Get`方法的描述参考其对应的`Set`*()方法。

表**7.10**

`GetRange()` `SetRange(minValue`，`maxValue)`：设置滑块的两端值。

`GetTickFreq()` `SetTickFreq(n`，`pos)`：使用参数n设置刻度的间隔。参数`pos`没有被使用，但是它仍然是必要的，将它设置为1。

`GetLineSize()` `SetLineSize(lineSize)`：设置你每按一下方向键，滑块所增加或减少的值。

`GetPageSize()` `SetPageSize(pageSize)`：设置你每按一下`PgUp`或`PgDn`键，滑块所增加或减少的值。

`GetValue()` `SetValue(value)`：设置滑块的值。

尽管滑块提供了一个可能范围内的值的快速的可视化的表示，但是它们也有两个缺点。其一是它们占据了许多的空间，另外就是使用鼠标精确地设置滑块是困难的。下面我们将讨论的微调控制器解决了上面的这两个问题。

如何得到那些灵巧的上下箭头按钮？

微调控制器是文本控件和一对箭头按钮的组合，它用于调整数字值，并且在你要求一个最小限度的屏幕空间的时候，它是替代滑块的最好选择。图7.8显示了`wxPython`的微调控制器控件。

图7.8

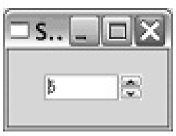


Figure 7.8
A spinner control
in wxPython

在`wxPython`中，类`wx.SpinCtrl`管理微调按钮和相应的文本显示。在接下来的部分，我们将创建一个微调控制器。

如何创建一个微调控制器

要使用 `wx.SpinCtrl` 来改变值，可通过按箭头按钮或通过在文本控件中输入。键入的非数字的文本将被忽略，尽管控件显示的是键入的非数字的文本。一个超出范围的值将被认作是相应的最大或最小值，尽管显示的是你输入的值。例7.8显示了 `wx.SpinCtrl` 的用法。

例**7.8** 使用 **`wx.SpinCtrl`**

```
import wx

class SpinnerFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Spinner Example',
                size=(100, 100))
        panel = wx.Panel(self, -1)
        sc = wx.SpinCtrl(panel, -1, "", (30, 20), (80, -1))
        sc.SetRange(1,100)
        sc.SetValue(5)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    SpinnerFrame().Show()
app.MainLoop()
```

几乎微调控件所有复杂的东西都是在其构造函数中，其构造函数如下：

`wx.SpinCtrl(parent` , `id` =-1, `value` = `wx.EmptyString` , `pos` = `wx.DefaultPosition` , `size` = `wx.DefaultSize` , `style` = `wx.SP_ARROW_KEYS` , `min` =0, `max` =100, `initial` =0, `name` =" `wxSpinCtrl` ")

参数 `value` 是虚设的。使用 `initial` 参数来设置该控件的值，并使用 `min` 和 `max` 来设置该控件的范围。

对于 `wx.SpinCtrl` 有两个样式标记。默认样式是 `wx.SP_ARROW_KEYS` ，它允许用户通过键盘上的上下箭头键来改变控件的值。样式 `wx.SP_WRAP` 使得控件中的值可以循环改变，也就是说你通过箭头按钮改变控件中的值到最大或最小值时，如果再继续，值将变为最小或最大，从一个极端到另一个极端。你也可以捕获 `EVT_SPINCTRL` 事件，它在当控件的值改变时产生（即使改变是直接由文本输入引起的）。如果文本改变了，将引发一个 `EVT_TEXT` 事件，就如同你使用一个单独的文本控件时一样。

如例7.8所示，你可以使用 `SetRange(minVal` , `maxVal)` 和 `SetValue(value)` 方法来设置范围和值。 `SetValue()` 函数要求一个字符串或一个整数。要得到值，使用方法： `GetValue()` （它返回一个整数）, `GetMin()` , 和 `GetMax()` 。

当你需要对微调控制器的行为有更多的控制时，如允许浮点数或一个字符串的列表，你可以把一个 `wx.SpinButton` 和一个 `wx.TextCtrl` 放到一起，并在它们之间建立一个联系。然后捕获来自 `wx.SpinButton` 的事件，并更

新 `wx.TextCtrl` 中的值。

如何生成一个进度条？

如果你只想图形化地显示一个数字值而不允许用户改变它，那么使用相应的 `wxPython` 窗口部件 `wx.Gauge` 。相关的例子就是图7.9所显示的进度条。

图**7.9**

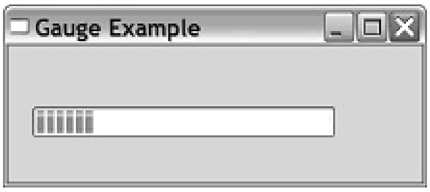


Figure 7.9 **A** `wx.Gauge` **displaying some progress**

例7.9显示了产生图7.9的代码。与本章中许多别的例子不同的是，这里我们增加了一个事件处理器。下面的代码在空闭时调整标尺的值，使得值周而复始的变化。

例**7.9** 显示并更新一个 **`wx.Gauge`**

```
import wx

class GaugeFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Gauge Example',
                size=(350, 150))
        panel = wx.Panel(self, -1)
        self.count = 0
        self.gauge = wx.Gauge(panel, -1, 50, (20, 50), (250, 25)
)
        self.gauge.SetBezelFace(3)
        self.gauge.SetShadowWidth(3)
        self.Bind(wx.EVT_IDLE, self.OnIdle)

    def OnIdle(self, event):
        self.count = self.count + 1
        if self.count  = 50:
            self.count = 0
        self.gauge.SetValue(self.count)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    GaugeFrame().Show()
app.MainLoop()
```

`wx.Gauge` 的构造函数类似于其它的数字的窗口部件：

wx.Gauge(parent , id , range , pos = wx.DefaultPosition , size = wx.DefaultSize , style = wx.GA_HORIZONTAL , validator = wx.DefaultValidator , name =" gauge ")

当你使用参数 range 来指定数字值时，该值代表标尺的上限，而下限总是0。默认样式 wx.GA_HORIZONTAL 提供了一个水平条。要将它旋转90度，使用 wx.GA_VERTICAL 样式。如果你是在 Windows 上，那么样式 wx.GA_PROGRESSBAR 给你的是来自 Windows 工具包的本地化的进度条。

作为一个只读控件， wx.Gauge 没有事件。然而，它的属性你可以设置。你可以使用 GetValue() , Set - Value(pos) , GetRange() ,和 SetRange(range) 来调整它的值和范围。如果你是在 Windows 上，并且没有使用本地进度条样式，那么你可以使用 SetBezelFace(width) and SetShadowWidth() 来改变3D效果的宽度。

给用户以选择

几乎每个应用程序都要求用户在一套预先定义的选项间进行选择。在 wxPython 中，有多种窗口部件帮助用户处理这种任务，包括复选框、单选按钮、列表框和组合框。接下来的部分将介绍这些窗口部件。

如何创建一个复选框？

复选框是一个带有文本标签的开关按钮。复选框通常成组的方式显示，但是每个复选框的开关状态是相互独立的。当你有一个或多个需要明确的开关状态的选项时，可以使用复选框。图7.10显示了一组复选框。

图**7.10**



Figure 7.10
A group of wxPython checkboxes

在 wxPython 中复选框很容易使用。它们是 wx.CheckBox 类的实例，并且通过把它们一起放入一个父容器中可以让它们在一起显示。例7.10提供了生成图7.10的代码。

例**7.10** 插入三个复选框到一个框架中

```
import wx

class CheckBoxFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Checkbox Example',
                size=(150, 200))
        panel = wx.Panel(self, -1)
        wx.CheckBox(panel, -1, "Alpha", (35, 40), (150, 20))
        wx.CheckBox(panel, -1, "Beta", (35, 60), (150, 20))
        wx.CheckBox(panel, -1, "Gamma", (35, 80), (150, 20))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    CheckBoxFrame().Show()
app.MainLoop()
```

`wx.CheckBox` 有一个典型的 `wxPython` 构造函数：

`wx.CheckBox(parent` , `id` , `label` , `pos` = `wx.DefaultPosition` , `size` = `wx.DefaultSize` , `style` =0, `name` =" `checkBox` ")

`label` 参数是复选框的标签文本。复选框没有样式标记，但是它们产生属于自己的独一无二的命令事件： `EVT_CHECKBOX` 。 `wx.CheckBox` 的开关状态可以使用 `GetValue()` 和 `SetValue(state)` 方法来访问,并且其值是一个布尔值。 `IsChecked()` 方法等同于 `GetValue()` 方法，只是为了让代码看起来更易明白。

如何创建一组单选按钮（radio `button` ）？

单选按钮是一种允许用户从几个选项中选择其一的窗口部件。与复选框不同，单选按钮是显式地成组配置，并且只能选择其中一个选项。当选择了新的选项时，上次的选择就关闭了。单选按钮的使用比复选框复杂些，因为它需要被组织到一组中以便使用。 `radio` `button` 的名字得自于老式轿车上有着同样行为的成组的选择按钮。

在 `wxPython` 中，有两种方法可以创建一组单选按钮。其一， `wx.RadioButton` ，它要求你一次创建一个按钮，而 `wx.RadioBox` 使你可以使用单一对象来配置完整的一组按钮，这些按钮显示在一个矩形中。

`wx.RadioButton` 类更简单些，在单选按钮对其它窗口部件有直接影响或单选按钮不是布置在一个单一的矩形中的情况下，它是首选。图7.11显示了一组 `wx.RadioButton` 对象的列子。

图**7.11**



Figure 7.11 Example of `wx.RadioButton` where radio buttons enable text control

我们在这个例子中使用 `wx.RadioButton` 的原因是因为每个单选按钮控制着一个关联的文本控件。由于窗口部件是位于这组单选按钮之外的，所以我们不能只用一个单选按钮框。

如何创建单选按钮

例7.11显示了图7.11的代码，它管理单选按钮和文本控件之间的联系。

例**7.11** 使用 **`wx.RadioButton`** 来控制另一个窗口部件

```
import wx

class RadioButtonFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Radio Example',
                size=(200, 200))
        panel = wx.Panel(self, -1)

#创建单选按钮
        radio1 = wx.RadioButton(panel, -1, "Elmo", pos=(20, 50),
 style=wx.RB_GROUP)
        radio2 = wx.RadioButton(panel, -1, "Ernie", pos=(20, 80)
)
        radio3 = wx.RadioButton(panel, -1, "Bert", pos=(20, 110)
)

#创建文本控件
        text1 = wx.TextCtrl(panel, -1, "", pos=(80, 50))
        text2 = wx.TextCtrl(panel, -1, "", pos=(80, 80))
        text3 = wx.TextCtrl(panel, -1, "", pos=(80, 110))
        self.texts = {"Elmo": text1, "Ernie": text2, "Bert": tex
t3}#连接按钮和文本
        for eachText in [text2, text3]:
            eachText.Enable(False)
        for eachRadio in [radio1, radio2, radio3]:#绑定事件
            self.Bind(wx.EVT_RADIOBUTTON, self.OnRadio, eachRadi
o)
        self.selectedText = text1

    def OnRadio(self, event):#事件处理器
        if self.selectedText:
            self.selectedText.Enable(False)
        radioSelected = event.GetEventObject()
        text = self.texts[radioSelected.GetLabel()]
        text.Enable(True)
        self.selectedText = text

if __name__ == '__main__':
    app = wx.PySimpleApp()
    RadioButtonFrame().Show()
app.MainLoop()
```

我们创建了单选按钮和文本框，然后使用字典来建立它们间的连接。一个 `for` 循环使得两个文本框无效，另一个 `for` 循环绑定单选按钮命令事件。当事件发生的时候，当前活动的文本框变为无效，与被敲击的按钮相匹配的文本框变为有效。

`wx.RadioButton` 的使用类似于是 `wx.CheckBox` 。它们的构造函数几乎是相同的，如下所示：

wx.RadioButton(parent , id , label , pos = wx.DefaultPosition , size = wx.DefaultSize , style =0, validator = wx.DefaultValidator , name =" radioButton ")

在复选框中， label 是相应按钮的显示标签。

wx.RB_GROUP 样式声明该按钮位于一组单选按钮开头。一组单选按钮的定义是很重要的，因为它控制开关行为。当组中的一个按钮被选中时，先前被选中的按钮被切换到未选中状态。在一个单选按钮使用 wx.RB_GROUP 被创建后，所有后来的被添加到相同父窗口部件中的单选按钮都被添加到同一组，直到另一单选按钮使用 wx.RB_GROUP 被创建，并开始下一个组。在例7.11中，第一个单选按钮是使用 wx.RB_GROUP 声明的，而后来的没有。结果导致所有的按钮都被认为在同一组中，这样一来，敲击它们中的一个时，先前被选中按钮将关闭。

使用单选框

通常，如果你想去显示一组按钮，分别声明它们不是最好的方法。取而代之， wxPython 使用 wx.RadioBox 类让你能够创建一个单一的对象，该对象包含了完整的组。如图7.12所示，它看起来非常类似一组单选按钮。

图**7.12**

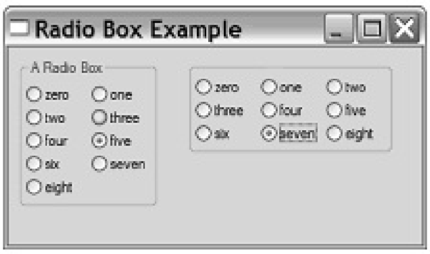


Figure 7.12 Two examples of wx.RadioBox built from the same underlying data with different configurations

要使用 wx.RadioBox 类，你所需要的全部就是构造函数。例7.12显示了图7.12的代码。

例**7.12** 建造单选框

```
import wx

class RadioBoxFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Radio Box Example',
                size=(350, 200))
        panel = wx.Panel(self, -1)
        sampleList = ['zero', 'one', 'two', 'three', 'four', 'fi
ve',
                      'six', 'seven', 'eight']
        wx.RadioBox(panel, -1, "A Radio Box", (10, 10), wx.Defau
ltSize,
                        sampleList, 2, wx.RA_SPECIFY_COLS)

        wx.RadioBox(panel, -1, "", (150, 10), wx.DefaultSize,
                        sampleList, 3, wx.RA_SPECIFY_COLS | wx.N
O_BORDER)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    RadioBoxFrame().Show()
app.MainLoop()
```

`wx.RadioBox` 的构造函数比简单的单选按钮更复杂，因为你需要去一下子为所有的按钮指定数据，如下所示：

```
wx.RadioBox(parent, id, label, pos=wx.DefaultPosition,
        size=wxDefaultSize, choices=None, majorDimension=0,
        style=wx.RA_SPECIFY_COLS, validator=wx.DefaultValidator,
        name="radioBox")
```

`label` 参数是静态文本，它显示在单选框的边框上。这些按钮使用 `choices` 参数指定，它是一个 `Python` 的字符串标签的序列。

如同网格的 `sizer` 一样，你通过使用规定一个维数的尺寸来指定 `wx.RadioBox` 的尺度， `wxPython` 在另一维度上自动填充。维度的主尺寸使用 `majorDimension` 参数指定。哪一维是主要的由样式标记决定。默认值是 `wx.RA_SPECIFY_COLS` 。在本例中，左框的列数被设置为2，右框的列数被设置为3，行数由 `choices` 列表中的元素数量动态的决定。如果你想得到相反的行为，你要将样式设置为 `wx.RA_SPECIFY_ROWS` 。如果你想在单选框被敲击时响应命令事件，那么这个命令事件是 `EVT_RADIOBOX` 。

`wx.RadioBox` 类有许多方法来管理框中的不同的单选按钮。这些方法使你能够处理一个特定的内部按钮，传递该按钮的索引。索引以0为开始，并按严格的顺序展开，它的顺序就是按钮标签传递给构造函数的顺序。表7.11列出了这些方法。

表7.11 `wx.RadioBox` 的方法

`EnableItem(n` , `flag)` : `flag` 参数是一个布尔值，它用于使索引为n的按钮有效或无效。要使整个框立即有效，使用 `Enable()` 。

`FindString(string)` ：根据给定的标签返回相关按钮的整数索引值，如果标签没有发现则返回-1。

`GetCount()` ：返回框中按钮的数量。

`GetItemLabel(n)` `SetItemLabel(n` , `string)` ：返回或设置索引为n的按钮的字符串标签。

`GetSelection()` `GetStringSelection()` `SetSelection(n)` `SetStringSelection(` `string)` : `GetSelection()` 和 `SetSelection()` 方法处理当前所选择的单选按钮的整数索引。 `GetStringSelection()` 返回当前所选择的按钮的字符串标签， `SetStringSelection()` 改变所选择的按钮的字符串标签为给定值。没有 `set` *()产生 `EVT_RADIOBOX` 事件。

`ShowItem(item` , `show)` : `show` 参数是一个布尔值，用于显示或隐藏索引为 `item` 的按钮。

单选按钮不是给用户一系列选择的唯一方法。列表框和组合框占用的空间也少，也可以被配置来让用户从同一组中作多个选择。

如何创建一个列表框？

列表框是提供给用户选择的另一机制。选项被放置在一个矩形的窗口中，用户可以选择一个或多个。列表框比单选按钮占据较少的空间，当选项的数目相对少的时候，列表框是一个好的选择。然而，如果用户必须将滚动条拉很远才能看到所有的选项的话，那么它的效用就有所下降了。图7.13显示了一个 `wxPython` 列表框。



Figure 7.13 A wx.ListBox. with a simple list of options

在 `wxPython` 中，列表框是类 `wx.ListBox` 的元素。该类的方法使你能够处理列表中的选择。

如何创建一个列表框

例7.13显示了产生图7.13的代码

例**7.13** 使用 **`wx.ListBox`**

```
import wx

class ListBoxFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'List Box Example',
                size=(250, 200))
        panel = wx.Panel(self, -1)

        sampleList = ['zero', 'one', 'two', 'three', 'four', 'fi
ve',
                      'six', 'seven', 'eight', 'nine', 'ten', 'e
leven',
                      'twelve', 'thirteen', 'fourteen']

listBox = wx.ListBox(panel, -1, (20, 20), (80, 120), sampleList,

                wx.LB_SINGLE)
        listBox.SetSelection(3)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ListBoxFrame().Show()
app.MainLoop()
```

`wx.ListBox`的构造函数类似于单选框的，如下所示：

`wx.ListBox(parent`, `id`, `pos` = `wx.DefaultPosition`, `size` = `wx.DefaultSize`, `choices` = `None`, `style` =0, `validator` = `wx.DefaultValidator`, `name` =" `listBox` ")

单选框和列表框的主要区别是`wx.ListBox`没有`label`属性。显示在列表中的元素放置在参数`choices`中，它是一个字符串的序列。列表框有三种互斥的样式，它决定用户如何从列表框中选择元素，说明在表7.12中。

用户通常对于多选有一些问题，因为它们一般希望见到的是单选列表，对于多选来说可能是有挑战性的（就像单选题和多选题一样），尤其是对于那些易受困扰的用户。如果你使用了一个多选的列表，我们建议你清楚地标明该列表。

表**7.12** 列表框的选择类型样式

`wx.LB_EXTENDED`：用户可以通过使用`shift`并敲击鼠标来选择一定范围内的连续的选项，或使用等同功能的按键。

`wx.LB_MULTIPLE`：用户可以一次选择多个选项（选项可以是不连续的）。实际上，在这种情况下，列表框的行为就像是一组复选框。

`wx.LB_SINGLE`：用户一次只能选一个选项。实际上，在这种情况下，列表框的行为就像是一组单选按钮。

有三种控制`wx.ListBox`中滚动条的显示的样式，如表7.13所示。

表**7.13** 列表框的滚动条类型样式

`wx.LB_ALWAYS_SB`：列表框将始终显示一个垂直的滚动条，不管有没有必要。

`wx.LB_HSCROLL`：如果本地控支持，那么列表框在选择项太多时，将创建一个水平滚动条。

`wx.LB_HSCROLL`：列表框只在需要的时候显示一个垂直的滚动条。这是默认样式。

还有一个样式`wx.LB_SORT`，它使得列表中的元素按字母顺序排序。

有两个专用于`wx.ListBox`的命令事件。`EVT_LISTBOX`事件在当列表中的一个元素被选择时触发（即使它是当前所选择的元素）。如果列表被双击，`EVT_LISTBOX_DCLICK`事件发生。

有一些专用于列表框的方法，你可以用来处理框中的项目。表7.14对许多的方法作了说明。列表框中的项目索引从0开始。

一旦你有了一个列表框，自然就想把它与其它的窗口部件结合起来使用，如下拉菜单，或复选框。在下一节，我们对此作讨论。

表**7.14** 列表框的方法

`Append(item)`：把字符串项目添加到列表框的尾部。

`Clear()`：清空列表框。

`Delete(n)`：删除列表框中索引为n的项目。

`Deselect(n)`：在多重选择列表框中，导致位于位置n的选项取消选中。在其它样式中不起作用。

`FindString(string)`：返回给定字符串的整数位置，如果没有发现则返回-1。

`GetCount()`：返回列表中字符串的数量。

`GetSelection()` `SetSelection(n`, `select)` `GetStringSelection()` `SetStringSelection(string`, `select)` `GetSelections()`：`GetSelection()`得到当前选择项的整数索引（仅对于单选列表）。对于多选列表，使用`GetSelections()`来返回包含所选项目的整数位置的元组。对于单选列表，`GetStringSelection()`返回当前选择的字符串。相应的`set`方法使用布尔值参数`select`设置指定字符串或索引选项的状态。使用这种方法改变选择不触发`EVT_LISTBOX`事件。

`GetString(n)` `SetString(n`, `string)`：得到或设置位置n处的字符串。

`InsertItems(items`, `pos)`：插入参数`items`中的字符串列表到该列表框中`pos`参数所指定的位置前。位置0表示把项目放在列表的开头。

`Selected(n)`：返回对应于索引为n的项目的选择状态的布尔值。

`Set(choices)`：重新使用`choices`的内容设置列表框。

如何合并复选框和列表框？

你可以使用类 `wx.CheckListBox` 来将复选框与列表框合并。图7.14显示了列表框和复选框在合并在一起的例子。

图**7.14**

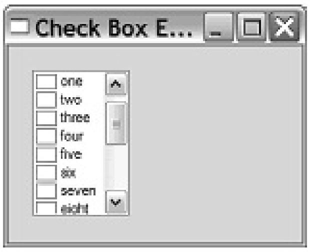


Figure 7.14 A check list box is very similar to a regular list box

`wx.CheckListBox` 的构造函数和大多数方法与 `wx.ListBox` 的相同。它有一个新的事件： `wx.EVT_CHECKLISTBOX` ，它在当列表中的一个复选框被敲击时触发。它有两个管理复选框的新的方法： `Check(n` , `check)` 设置索引为n的项目的选择状态， `IsChecked(item)` 在给定的索引的项目是选中状态时返回 `True` 。

如果我想要下拉形式的选择该怎么做？

下拉式选择是一种仅当下拉箭头被敲击时才显示选项的选择机制。它是显示所选元素的最简洁的方法，当屏幕空间很有限的时候，它是最有用的。图7.15显示了一个关闭的下拉式选择。图7.16显示了一个打开的下拉式选择。

图**7.15**



Figure 7.15 A pull-down choice, with no selection

图**7.16**

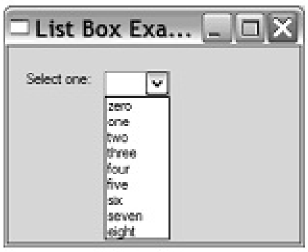


Figure 7.16 A pull-down choice in the process of having an element selected

下拉式选择的使用与标准的列表框是很相似的。例7.14显示了如何创建一个下拉式选择。

例**7.14**

```
import wx

class ChoiceFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Choice Example',
                size=(250, 200))
        panel = wx.Panel(self, -1)
        sampleList = ['zero', 'one', 'two', 'three', 'four', 'fi
ve',
                      'six', 'seven', 'eight']
        wx.StaticText(panel, -1, "Select one:", (15, 20))
        wx.Choice(panel, -1, (85, 18), choices=sampleList)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ChoiceFrame().Show()
app.MainLoop()
```

`wx.Choice` 的构造函数与列表框的基本相同：

`wx.Choice(parent` , `id` , `pos` = `wx.DefaultPosition` , `size` = `wx.DefaultSize` , `choices` = `None` , `style` =0, `validator` = `wx.DefaultValidator` , `name` =" `choice` ")

`wx.Choice` 没有专门的样式，但是它有独特的命令事件： `EVT_CHOICE` 。几乎表7.14中所有适用于单选列表框的方法都适用于 `wx.Choice` 对象。

我能够将文本域与列表合并在一起吗？

将文本域与列表合并在一起的窗口部件称为组合框，其本质上是一个下拉选择和文本框的组合。图7.17显示了一个组合框。

图**7.17** 左边是 `wx.CB_DropDOWN` 样式，右边是 `wx.CB_SIMPLE` 样式

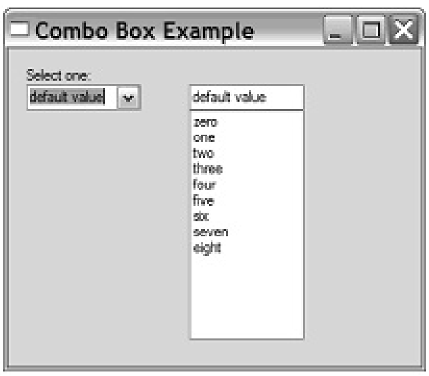


Figure 7.17 A combo box showing the left box in the style wx.CB_DROPDOWN, and the right in wx.CB_SIMPLE

在 `Windows` 上，你可以使用右边的样式，它是一个列表框和文本框的组合。

创建组合框的代码与我们已经见过的选择是类似的。该类是 `wx.ComboBox` ，它是 `wx.Choice` 的一个子类。例7.15显示了图7.17的代码：

例**7.15**

```
import wx

class ComboBoxFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Combo Box Example',
                size=(350, 300))
        panel = wx.Panel(self, -1)
        sampleList = ['zero', 'one', 'two', 'three', 'four', 'fi
ve',
                      'six', 'seven', 'eight']
        wx.StaticText(panel, -1, "Select one:", (15, 15))
        wx.ComboBox(panel, -1, "default value", (15, 30), wx.Def
aultSize,
                    sampleList, wx.CB_DropDOWN)
        wx.ComboBox(panel, -1, "default value", (150, 30), wx.De
faultSize,
                        sampleList, wx.CB_SIMPLE)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ComboBoxFrame().Show()
app.MainLoop()
```

`wx.ComboBox` 的构造函数如下所示：

```
wx.ComboBox(parent, id, value="", pos=wx.DefaultPosition,
        size=wx.DefaultSize, choices, style=0,
        validator=wx.DefaultValidator, name="comboBox")
```

对于 `wx.ComboBox` 来说有4种样式。其中的两种决定了如何绘制组合框： `wx.CB_DropDOWN` 创建一个带有下拉列表的组合框， `wx.CB_SIMPLE` 创建一个带有列表框的组合框。在 `Windows` 上你可以只使用 `wx.CB_SIMPLE` 样式。任何组合框都可以被指定为 `wx.CB_READONLY` 样式，它防止用户在文本域中键入。当组合框被指定为只读时，所做的选择必须来自于选择列表的元素之一，即使你用程序来设置它也不行。最后 `wx.CB_SORT` 样式导致选择列表中的元素按字母顺序显示。

由于 `wx.ComboBox` 是 `wx.Choice` 的子类，所有的 `wx.Choice` 的方法都能被组合框调用，如表7.14所示。另外，还有许多方法被定义来处理文本组件，它们的行为同 `wx.TextCtrl` （参见表7.4)，所定义的方法有 `Copy()` , `Cut()` ,

`GetInsertionPoint()` , `GetValue()` , `Paste()` , `Replace(from` , `to` , `text)` , `Remove(from` , `to)` , `SetInsertionPoint(pos)` , `SetInsertionPointEnd()` ,和 `SetValue()` 。

本章小结

在这一章中，我们给你展示了如何使用 `wxPython` 中许多最基本和常用的控件。这些通用的版本在跨平台使用时显得一致性较好。

1、对于静态文本标签的显示，你可以使用 `wx.StaticText` 类。还有一个完全用 `wxPython` 实现的版本，名为 `wx.lib.stattext.GenStaticText` 。

2、如果你需要一个控件以让用户输入文本，那么使用类 `wx.TextCtrl` 。它允许单行和多行的输入，还有密码掩饰和其它的功用。如果本地控支持它，你可以使用 `wx.TextCtrl` 来得到样式文本。样式是 `wx.Text` - `Attr` 类的实例， `wx.Font` 包含字体信息。对于所有的系统，你可以使用类 `wx.stc.StyledTextCtrl` （它是 `wxPython` 对开源 `Scintilla` 文本组件的封装）在一个可编辑的文本组件中实现颜色和字体样式。

3、创建按钮，使用 `wx.Button` 类，它也有一个通用版 `wx.lib.buttons.GenButton` 。按钮可以使用位图来代替一个文本标签（ `wx.BitmapButton` ），或在按下和未按下之间有一个开关状态。还有一个等价于位图和开关按钮的通用版，它比标准版有更全面的特性。

4、有一些方法用于选择或显示数字值。你可以使用 `wx.Slider` 类来显示一个垂直或水平的滑块。 `wx.SpinCtrl` 显示一个可以使用上下按钮来改变数字值的文本控件。 `wx.Gauge` 控件显示一个进度条指示器。

5、你可以从一系列的控件中选出让用户从列表选项作出选择的最佳控件，最佳控件所应考虑的条件是选项的数量，用户能否多选和你想使用的屏幕空间的总量。复选框使用 `wx.CheckBox` 类。这儿有两个方法去得到单选按钮： `wx.RadioButton` 给出单个单选按钮，而 `wx.RadioBox` 给出显示在一起的一组按钮。这儿有几个列表显示控件，它们的用法相似。列表框的创建使用 `wx.ListBox` ，并且你可以使用 `wx.CheckListBox` 来增加复选框。对于更简洁的下拉式，使用 `wx.Choice.` `wx.ComboBox` 合并了列表和文本控件的特性。

到目前为止，我们已经涉及了基本的常用窗口部件，在接下来的章节，我们将讨论不同种类的框架，它可以用这些框架来包含我们已经涉及了基本的常用窗口部件。

第八章 将构件放入窗体中

在你的wxPython中，所有的用户交互行为都发生在一个窗口部件容器中，它通常被称作窗口，在wxPython 中被称为框架。在这一章中，我们将讨论wxPython中的几个不同样式的框架。这个主要的wx.Frame有几个不同的框架样式，这些样式可以改变wx.Frame的外观。另外，wxPython提供了小型框架和实现多文档界面的框架。框架可以使用分隔条来划分为不同的部分，并且可以通过滚动条的使用来包含比框架本身大的面板（panel）。

框架的寿命

我们将通过讨论框架最基本的元素：创建和除去它们，来作为我们的开始。创建框架包括了解可以应用的所有样式元素；框架的去除可能比你原本想像的要复杂。

如何创建一个框架？

在本书中我们已经见过了许多的框架创建的例子，但是我们仍将再回顾一下框架创建的初步原则。

创建一个简单的框架

框架是类wx.Frame的实例。例8.1显示了一个非常简单的框架创建的例子。

例**8.1** 创建基本的**wx.Frame**

```
import wx

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = wx.Frame(None, -1, "A Frame", style=wx.DEFAULT_FRAME
_STYLE,
        size=(200, 100))
    frame.Show()
    app.MainLoop()
```

上面的代码创建一个带有标题的框架，其大小是(200,100)。表8.1中的默认样式提供了标准框架的装饰如关闭框、最小化和最大化框。结果如图8.1所示。

图8.1



Figure 8.1
The simple frame

wx.Frame的构造函数类似于我们在第7章见到的其它窗口部件的构造函数：

```
wx.Frame(parent, id=-1, title="", pos=wx.DefaultPosition,
        size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE,
        name="frame")
```

这里有超过十余种之多的专用于wx.Frame的样式标记，我们将在下一部分涵盖它们。默认样式为你提供了最小化和最大化框、系统下拉菜单、可调整尺寸的粗边框和一个标题。

这里没有与一个wx.Frame挂钩的事件类型。但是，由于一个wx.Frame是你的屏幕上用户最可能去关闭的元素，所以你通常想去为关闭事件定义一个处理器，以便子窗口和数据被妥善的处理。

创建框架的子类

你将很少直接创建wx.Frame的实例。正如我们在本书中所见过的其它例子一样，一个典型的wxPython应用程序创建wx.Frame的子类并创建那些子类的实例。这是因为wx.Frame的独特的情形——虽然它自身定义了很少的行为，但是带有独自的初始化程序的子类是放置有关你的框架的布局和行为的最合理的地方。不创建子类而构造你应用程序的特定的布局是有可能，但除了最简单的应用程序以外，那是不容易的。例8.2展示了wx.Frame子类的例子。

例**8.2** 一个简单的框架子类

```
import wx

class SubclassFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Frame Subclass',
                size=(300, 100))
        panel = wx.Panel(self, -1)
        button = wx.Button(panel, -1, "Close Me", pos=(15, 15))
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

    def OnCloseMe(self, event):
        self.Close(True)

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    SubclassFrame().Show()
app.MainLoop()
```

运行结果如图8.2所示

图8.2



Figure 8.2 The simple frame as a subclass

我们在许多其它的例子中已经见过了这种基本的结构，因此让我们来讨论上面代码中特定于框架的部分。wx.Frame.init方法与wx.Frame构造函数有相同的信息。子类自身的构造器除了self没有其它参数，它允许你作为程序员去定义参数，所定义的参数将传递给其父类，并且使你可以不用重复指定与父类相同的参数。

同样值得注意的是，框架的子窗口部件被放置在一个面板（panel）中。面板（panel)是类wx.Panel的实例，它是其它有较少功能的窗口部件的容器。你基本上应该使用一个wx.Panel作为你的框架的顶级子窗口部件。有一件事情就是，多层次的构造可以使得更多的代码能够重用，如相同的面板和布局可以被用于多个框架中。在框架中使用wx.Panel给了你一些对话框的功能。这些功能以成对的方式表现。其一是，在MS Windows操作系统下，wx.Panel实例的默认背景色以白色代替了灰色。其二，面板（panel）可以有一个默认的项目，该项目在当回车键被按下时自动激活，并且面板（panel）以与对话框大致相同的办法响应tab键盘事件，以改变或选择默认项目。

有些什么不同的框架样式？

wx.Frame有许多的可能的样式标记。通常，默认样式就是你想要的，但也有一些有用的变种。我们将讨论的第一组样式控制框架的形状和尺寸。尽管不是强制性的，但是这些标记应该被认为是互斥的——一个给定的框架应该只使用它们中的一个。表8.1说明了形状和尺寸标记。

表**8.1** 框架的形状和尺寸标记

wx.FRAME_NO_TASKBAR	一个完全标准的框架，除了一件事：在Windows系统和别的支持这个特性的系统下，它不显示在任务栏中。当最小化时，该框架图标化到桌面而非任务栏。
wx.FRAME_SHAPED	非矩形的框架。框架的确切形状使用SetShape()方法来设置。窗口的形状将在本章后面部分讨论。
wx.FRAME_TOOL_WINDOW	该框架的标题栏比标准的小些，通常用于包含多种工具按钮的辅助框架。在Windows操作系统下，工具窗口将不显示在任务栏中。
wx.ICONIZE	窗口初始时将被最小化显示。这个样式仅在Windows系统中起作用。
wx.MAXIMIZE	窗口初始时将被最大化显示（全屏）。这个样式仅在Windows系统中起作用。
wx.MINIMIZE	同wx.ICONIZE。

上面这组样式中，屏幕画面最需要的样式是wx.FRAME_TOOL_WINDOW。图8.3显示了一个小的结合使用了wx.FRAME_TOOL_WINDOW、wx.CAPTION和wx.SYSTEM_MENU样式的例子。

图**8.3**



Figure 8.3 An example of the tool window style

这里有两个互斥的样式，它们控制一个框架是否位于别的框架的上面，无论别的框架是否获得了焦点。这对于那些小的不是始终可见的对话框是有用的。表8.2说明了这两个样式。最后，这还有一些用于放置在你的窗口上的装饰。如果你没有使用默认样式，那么这些装饰将不被自动放置到你的窗口上，你必须添加它们，否则容易导致窗口不能关闭或移动。表8.3给出了这些装饰的列表。

表**8.2** 针对窗口漂浮行为的样式

wx.FRAME_FLOAT_ON_PARENT	框架将漂浮在其父窗口（仅其父窗口）的上面。（很明显，要使用这个样式，框架需要有一个父窗口）。其它的框架可以遮盖这个框架。
wx.STAY_ON_TOP	该框架将始终在系统中其它框架的上面。（如果你有多个框架使用了这个样式，那么它们将相互重叠，但对于系统中其它的框架，它们仍在上面。）

默认的样式wx.DEFAULT_FRAME_STYLE等同于wx.MINIMIZE_BOX | wx.MAXIMIZE_BOX | wx.CLOSE_BOX | wx.RESIZE_BORDER | wx.SYSTEM_MENU |wx.CAPTION。这个样式创建了一个典型的窗口，你可以调整大小，最小化，最大化，或关闭。一个很好的主意就是当你想要使用除默认样式以外的样式时，将默认样式与其它的样式组合在一起，以确保你有正确的一套装饰。例如，要创建一个工具框架，你可以使用style=wx.DEFAULT_FRAME_STYLE | wx.FRAME_TOOL_WINDOW。记住，你可以使用^操作符来去掉不要的样式。

表**8.3** 用于装饰窗口的样式

wx.CAPTION	给窗口一个标题栏。如果你要放置最大化框、最小化框、系统菜单和上下文帮助，那么你必须包括该样式。
wx.FRAME_EX_CONTEXTHELP	这是用于Windows操作系统的，它在标题栏的右角放置问号帮助图标。这个样式是与wx.MAXIMIZE_BOX和WX.MINIMIZE_BOX样式互斥的。它是一个扩展的样式，并且必须使用两步来创建，稍后说明。
wx.FRAME_EX_METAL	在Mac OS X上，使用这个样式的框架有一个金属质感的外观。这是一个附加样式，必须使用SetExtraStyle方法来设置。
wx.MAXIMIZE_BOX	在标题栏的标准位置放置一个最大化框。
wx.MINIMIZE_BOX	在标题栏的标准位置放置一个最小化框。
wx.CLOSE_BOX	在标题栏的标准位置放置一个关闭框。
wx.RESIZE_BORDER	给框架一个标准的可以手动调整尺寸的边框。
wx.SIMPLE_BORDER	给框架一个最简单的边框，不能调整尺寸，没有其它装饰。该样式与所有其它装饰样式是互斥的
wx.SYSTEM_MENU	在标题栏上放置一个系统菜单。这个系统菜单的内容与你所使用的装饰样式有关。例如，如果你使用wx.MINIMIZE_BOX，那么系统菜单项就有“最小化”选项。

如何创建一个有额外样式信息的框架？

wx.FRAME_EX_CONTEXTHELP是一个扩展样式，意思是样式标记的值太大以致于不能使用通常的构造函数来设置（因为底层C++变量类型的特殊限制）。通常你可以在窗口部件被创建后，使用SetExtraStyle方法来设置额外的样式，但是某些样式，比如wx.FRAME_EX_CONTEXTHELP，必须在本地UI（用户界面）对象被创建之前被设置。在wxPython中，这需要使用稍微笨拙的方法来完成，即分两步构建。之后标题栏中带有我们熟悉的问号图标的框架就被创建了。如图8.4所示。

图8.4



Figure 8.4 A frame with the extended context help enabled

标记值必须使用SetExtraStyle()方法来设置。有时，额外样式信息必须在框架被实例化前被设置，这就导致了一个问题：你如何对于一个不存在的实例调用一个方法？在接下来的部分，我们将展示实现这种操作的两个机制。

添加额外样式信息

在wxPython中，额外样式信息在创建之前通过使用专门的类wx.PreFrame来被添加，它是框架的一种局部实例。你可以在预框架（preframe）上设置额外样式位，然后使用这个预框架（preframe）来创建实际的框架。例8.3显示了在一个子类的构造器中如何完成这两步（two-step）的构建。注意，在wxPython中它实际上是三步（在C++ wxWidgets工具包中，它是两步（two-step），我们只是沿用这个叫法而已）。

例**8.3**

```
import wx

class HelpFrame(wx.Frame):

    def __init__(self):
        pre = wx.PreFrame() #1 预构建对象
        pre.SetExtraStyle(wx.FRAME_EX_CONTEXTHELP)
        pre.Create(None, -1, "Help Context", size=(300, 100),
                style=wx.DEFAULT_FRAME_STYLE ^
                (wx.MINIMIZE_BOX | wx.MAXIMIZE_BOX)) #2 创建框架
        self.PostCreate(pre) #3 底层C++指针的传递

if __name__ == '__main__':
    app = wx.PySimpleApp()
    HelpFrame().Show()
    app.MainLoop()
```

#1 创建wx.PreFrame()的一个实例（关于对话框，这有一个类似的wx.PreDialog()——其它的wxWidgets窗口部件有它们自己的预类）。在这个调用之后，你可以做你需要的其它初始化工作。

#2 调用Create()方法创建框架。

#3 这是特定于wxPython的，并且不由C++完成。PostCreate方法做一些内部的内务处理，它实例化一个你在第一步中创建的封装了C++的对象。

添加额外样式信息的通用方法

先前的算法有点笨拙，但是它可以被重构得容易一点，以便于管理维护。第一步是创建一个公用函数，它可以管理任何分两步的创建。例8.4提供了一个例子，它使用Python的内省性能来调用以变量形式被传递的函数。这个例子用于在Python的一个新的框架实例化期间的init方法中被调用。

例**8.4** 一个公用的两步式创建函数

```
def twoStepCreate(instance, preClass, preInitFunc, *args,**kwarg
s):
        pre = preClass()
        preInitFunc(pre)
        pre.Create(*args, **kwargs)
        instance.PostCreate(pre)
```

在例8.4中，函数要求三个必须的参数。instance参数是实际被创建的实例。preClass参数是临时的预类的类对象——对框架预类是wx.PreFrame。preInitFunc是一个函数对象，它通常作为回调函数用于该实例的初始化。这三个参数之后，我们可以再增加任意数量的其它可选参数。

这个函数的第一行，pre = preClass()，内省地实例化这个预创建对象，使用作为参数传递过来的类对象。下面一行根据参数preInitFunc内省地调用回调函数，它通常设置扩展样式标记。然后pre.Create()方法被调用，它使用了可选的参数。最后，PostCreate方法被调用来将内在的值从pre移给实例。至此，instance参数已经完全被创建了。假设twoStepCreate已被导入，那么上面的公用函数可以如例8.5被使用。

例**8.5** 另一个两步式的创建，使用了公用函数

```
import wx

class HelpFrame(wx.Frame):

def __init__(self, parent, ID, title,pos=wx.DefaultPosition, siz
e=(100,100),style=wx.DEFAULT_DIALOG_STYLE):
        twoStepCreate(self, wx.PreFrame, self.preInit, parent,
        id, title, pos, size, style)

def preInit(self, pre):
        pre.SetExtraStyle(wx.FRAME_EX_CONTEXTHELP)
```

类wx.PreFrame和函数self.preInit被传递给公用函数，并且preInit方法被定义为回调函数。

当关闭一个框架时都发生了什么？

当你关闭一个框架时，它最终消失了。除非这个框架被明确地告诉不关闭。换句话说，那关闭不是直接了当的。在wxPython的窗口部件关闭体系之后的用意是，给正在关闭的窗口部件充足的机会来关闭或释放它所占用任何非wxPython资源。如果你占用了某种昂贵的外部资源，如一个大的数据结构或一个数据库连接，那么该意图是特别受欢迎的。

诚然，在C++ wxWidgets世界里，由于C++不为你管理内在分配的清理工作，管理资源是更严肃的问题。在wxPython中，对于多步的关闭过程的显式需求就很少，但它对于在过程中使用额外的钩子仍然是有用的。（随便说一下，我们在这一节中从

单词"框架"切换到单词"窗口部件"是故意的——因为在本节中的所有内容都适用于所有顶级窗口部件，如框架或对话框）。

何时用户触发关闭过程

关闭过程最常由用户触发，如敲击一个关闭框或选择系统菜单中的关闭项或当应用程序响应其它某个事件而调用框架的Close方法。当上述情况发生时，wxPython架构引发一个EVT_CLOSE事件。像wxPython 架构中的其它别的事件一样，你可以在绑定一个事件处理器以便一个EVT_CLOSE事件发生时调用。

如果你不声明你自己的事件处理器，那么默认的行为将被调用。默认的行为对于框架和对话框是不同的。

1、默认情况下，框架处理器调用Destroy()方法并删除该框架和它的所有的组件。

2、默认情况下，对话框的关闭处理器不销毁该对话框——它仅仅模拟取消按钮的按下，并隐藏对话框。该对话框对象仍继续存在在内存中。因此，如果需要的话，应用程序可以从它的数据输入部件获取值。当应用程序完成了对对话框的使用后，应该调用对话框的Destroy()方法。

如果你编写你自己的关闭处理器，那么你可以使用它来关闭或删除任何外部的资源，但是，如果你选择去删除框架的话，显式地调用Destroy()方法是你的责任。尽管Destroy()经常被Close()调用，但是只调用Close()方法不能保证框架的销毁。在一定的情形下，决定不销毁框架是完全可以的，如当用户取消了关闭。然而，你仍然需要一个方法来销毁该框架。如果你选择不去销毁窗口，那么调用关闭事件的wx.CloseEvent.Veto()方法来通知相关部分：框架拒绝关闭，是一个好的习惯。

如果你选择在你的程序的别处而非关闭处理器中关闭你的框架，例如从一个不同的用户事件像一个菜单项，那么我们建议使用的机制是调用框架的Close()方法。这将启动一个和系统关闭行为相同的过程。如果你要确保框架一定被删除，那么你可以直接调用Destroy()方法；然而，如果你这样做了，可能会导致框架所管理的资源或数据没有被释放或保存。

什么时候系统触发关闭过程

如果关闭事件是由系统自己触发的，对于系统关闭或类似情况，你也有一种办法管理该事件。wx.App 类接受一个EVT_QUERY_END_SESSION事件，如果需要的话，该事件使你能够否决应用程序的关闭，如果所有运行的应用已经批准了系统或GUI环境的关闭的话，那么随后会有一个EVT_END_SESSION事件。你选择去否决关闭的行为是与依赖于具体系统的。

最后，值得注意的是，调用一个窗口部件的Destroy()方法不意味该部件被立即销毁。销毁实际上是当事件循环在未来空闲时（任何未被处理的事件被处理之后）才被处理的。这就防止了处理已不存在的窗口部件的事件。

在接下来的两节，我们的讨论将从一个框架的生命周期切换到在框架生命周期里，你能够用框架来做些什么。

使用框架

框架包含了许多方法和属性。其中最重要的是那些查找框架中任意窗口部件的方法，和滚动框架中内容的方法。在这一节，我们将讨论如何实现这些。

wx.Frame有那些方法和属性？

这部分中的表包含了wx.Frame和它的父类wx.Window的最基本的属性。这些属性和方法的许多在本书中的其它地方有更详细的说明。表8.4包含了wx.Frame的一些公共的可读、可修改的属性。

表**8.4 wx.Frame**的公共属性

GetBackgroundColor(),SetBackgroundColor(wx.Color)	背景色是框架中没有被其子窗口部件覆盖住的那些部分的颜色。你可以传递一个wx.Color或颜色名给设置方法。任何传递给需要颜色的wxPython方法的字符串，都被解释为对函数wx.NamedColour()的调用。
GetId(),SetId(int)	返回或设置窗口部件的标识符。
GetMenuBar(),SetMenuBar(wx.MenuBar)	得到或设置框架当前使用的的菜单栏对象，如果没有菜单栏，则返回None。
GetPosition(),GetPositionTuple(),SetPosition(wx.Point)	以一个wx.Point或Python元组的形式返回窗口左上角的x,y的位置。对于顶级窗口，该位置是相对于显示区域的坐标，对于子窗口，该位置是相对于父窗口的坐标。

GetSize() GetSizeTuple() SetSize(wx.Size)：C++版的get或*set*方法被覆盖。默认的get或*set*使用一个wx.Size对象。GetSizeTuple()方法以一个Python元组的形式返回尺寸。也可以参看访问该信息的另外的方法SetDimensions()。

GetTitle() SetTitle(String)：得到或设置框架标题栏的字符串。

表8.5显示了一些wx.Frame的非属性类的更有用的方法。其中要牢记的是Refresh()，你可以用它来手动触发框架的重绘。

表8.5 wx.Frame的方法

Center(direction=wx.BOTH)：框架居中（注意，非美语的拼写Centre，也被定义了的）。参数的默认值是wx.BoTH，在此情况下，框是在两个方向都居中的。参数的值若是wx.HORIZONTAL或wx.VERTICAL，表示在水平或垂直方向居中。

Enable(enable=true)：如果参数为true，则框架能够接受用户的输入。如果参数为False，则用户不能在框架中输入。相对应的方法是Disable()。

GetBestSize()：对于wx.Frame，它返回框架能容纳所有子窗口的最小尺寸。

Iconize(iconize)：如果参数为true，最小化该框架为一个图标（当然，具体的行为与系统有关）。如果参数为False，图标化的框架恢复到正常状态。

IsEnabled()：如果框架当前有效，则返回True。

IsFullScreen()：如果框架是以全屏模式显示的，则返回True，否则False。细节参看ShowFullScreen。

IsIconized()：如果框架当前最小化为图标了，则返回True，否则False。

IsMaximized()：如果框架当前是最大化状态，则返回True，否则False。

IsShown()：如果框架当前可见，则返回True。

IsTopLevel()：对于顶级窗口部件如框架或对话框，总是返回True，对于其它类型的窗口部件返回False。

Maximize(maximize)：如果参数为True，最大化框架以填充屏幕（具体的行为与系统有关）。这与敲击框架的最大化按钮所做的相同，这通常放大框架以填充桌面，但是任务栏和其它系统组件仍然可见。

Refresh(eraseBackground=True, rect=None)：触发该框架的重绘事件。如果rect是none，那么整个框架被重画。如果指定了一个矩形区域，那么仅那个矩形区域被重画。如果eraseBackground为True，那么这个窗口的北影也将被重画，如果为False，那么背景将不被重画。

SetDimensions(x, y, width, height, sizeFlags=wx.SIZE_AUTO)：使你能够在一个方法调用中设置窗口的尺寸和位置。位置由参数x和y决定，尺寸由参数width和height决定。前四个参数中，如果有的为-1，那么这个-1将根据参数sizeFlags的值作相应的解释。表8.6包含了参数sizeFlags的可能取值。

Show(show=True)：如果参数值为True，导致框架被显示。如果参数值为False，导致框架被隐藏。Show(False)等同于Hide()。

ShowFullScreen(show, style=wx.FULLSCREEN_ALL)：如果布尔参数是True，那么框架以全屏的模式被显示——意味着框架被放大到填充整个显示区域，包括桌面上的任务栏和其它系统组件。如果参数是False，那么框架恢复到正常尺寸。style参数是一个位掩码。默认值wx.FULLSCREEN_ALL指示wxPython当全屏模式时隐藏所有窗口的所有样式元素。后面的这些值可以通过使用按位运算符来组合，以取

消全屏模式框架的部分装饰：wx.FULLSCREEN_NOBORDER, wx.FULLSCREEN_NOCAPTION, wx.FULLSCREEN_NOMENUBAR, wx.FULLSCREEN_NOSTATUSBAR, wx.FULLSCREEN_NOTOOLBAR。

表8.5中说明的SetDimensions()方法在用户将一个尺寸指定为-1时，使用尺寸标记的一个位掩码来决定默认行为。表8.6说明了这些标记。

这些方法没有涉及框架所包含的孩子的位置问题。这个问题要求框架的孩子自已去说明它。

表**8.6** 关于**SetDimensions**方法的尺寸标记

wx.ALLOW_MINUS_ONE：一个有效的位置或尺寸。

wx.SIZE_AUTO：转换为一个wxPython默认值。

wx.SIZE_AUTO_HEIGHT：一个有效的高度，或一个wxPython默认高度。

wx.SIZE_AUTO_WIDTH：一个有效的宽度，或一个wxPython默认宽度。

wx.SIZE_USE_EXISTING：使用现有的尺寸。

如何查找框架的子窗口部件？

有时候，你将需要查找框架或面板(panel)上的一个特定的窗口部件，并且你没有它的相关引用。如第6章所示的这种情况的一个公用的应用程序，它查找与所选菜单相关的实际的菜单项对象（因为事件不包含对它的一个引用）。另一种情况就是，当你想基于一个项的事件去改变系统中其它任一窗口部件的状态时。例如，你可能有一个按钮和一个菜单项，它们互相改变彼此的开关状态。当按钮被敲击时，你需要去得到菜单项以触发它。例8.6显示了一个摘自第7章的一个小的例子。在这个代码中，FindItemById()方法用来去获得与事件对象所提供的ID相关的菜单项。该项的标签被用来驱动所要求的颜色的改变。

例**8.6** 通过**ID**查找项目的函数

```
def OnColor(self, event):
        menubar = self.GetMenuBar()
        itemId = event.GetId()
        item = menubar.FindItemById(itemId)
        color = item.GetLabel()
        self.sketch.SetColor(color)
```

在wxPython中，有三种查找子窗口部件的方法，它们的行为都很相似。这些方法对任何作为容器的窗口部件都是适用的，不单单是框架，还有对话框和面板(panel)。你可以通过内部的wxPython ID查寻一个窗口部件，或通过传递给构造函数的名字（在name参数中），或通过文本标签来查寻。文本标签被定义为相应窗口部件的标题，如按钮和框架。

这三种方法是：

- 1.wx.FindWindowById(id, parent=None)

  1.wx.FindWindowByName(name, parent=None)

  1.wx.FindWindowByLabel(label, parent=None)

这三种情况中，parent参数可以被用来限制为对一个特殊子层次的搜索（也就是，它等同于父类的Find方法）。还有，FindWindowByName()首先按name参数查找，如果没有发现匹配的，它就调用FindWindowByLabel()去查找一个匹配。

如何创建一个带有滚动条的框架？

在wxPython中，滚动条不是框架本身的一个元素，而是被类wx.ScrolledWindow控制。你可以在任何你要使用wx.Panel的地方使用wx.ScrolledWindow，并且滚动条移动所有在滚动窗口中的项目。图8.5和图8.6显示了滚动条，包括它的初始状态和滚动后的状态。从图8.5到图8.6，左上的按钮移出了视野，右下的按钮移进了视野。

在这一节，我们将讨论如何去创建一个带有滚动条的窗口以及如何在你的程序中处理滚动行为。

图**8.5**

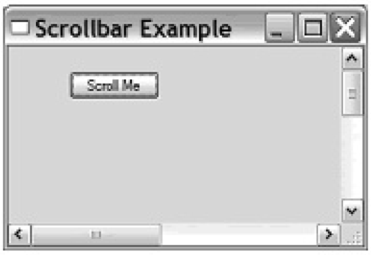


Figure8.5 Awx.Scrolled-Window after initial creation

图**8.6**

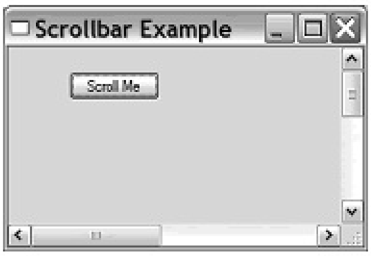


Figure8.5 Awx.Scrolled-Window after initial creation

如何创建滚动条

例8.7显示了用于创建滚动窗口的代码。

例**8.7** 创建一个简单的滚动窗口

```
import wx

class ScrollbarFrame(wx.Frame):
    def __init__(self):
wx.Frame.__init__(self, None, -1, 'Scrollbar Example',
                size=(300, 200))
        self.scroll = wx.ScrolledWindow(self, -1)
        self.scroll.SetScrollbars(1, 1, 600, 400)
        self.button = wx.Button(self.scroll, -1, "Scroll Me", po
s=(50, 20))
        self.Bind(wx.EVT_BUTTON,  self.OnClickTop, self.button)
        self.button2 = wx.Button(self.scroll, -1, "Scroll Back",
 pos=(500, 350))
        self.Bind(wx.EVT_BUTTON, self.OnClickBottom, self.button
2)

    def OnClickTop(self, event):
        self.scroll.Scroll(600, 400)

    def OnClickBottom(self, event):
        self.scroll.Scroll(1, 1)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ScrollbarFrame()
    frame.Show()
    app.MainLoop()
```

wx.ScrolledWindow的构造函数几乎与wx.Panel的相同：

```
wx.ScrolledWindow(parent, id=-1, pos=wx.DefaultPosition,
        size=wx.DefaultSize, style=wx.HSCROLL | wx.VSCROLL,
        name="scrolledWindow")
```

所有的这些属性的行为都如你所愿，尽管size属性是它的父亲中的面板的物理尺寸，而非滚动窗口的逻辑尺寸。

指定滚动区域的尺寸

有几个自动指定滚动区域尺寸的方法。手工指定最多的方法如例8.1所示，使用了方法SetScrollBars：

```
SetScrollbars(pixelsPerUnitX, pixelsPerUnitY, noUnitsX, noUnitsY
,
        xPos=0, yPos=0, noRefresh=False)
```

其中关键的概念是滚动单位，它是滚动条的一次移动所引起的窗口中的转移距离。前面的两个参数pixelsPerUnitX和PixelsPerUnitY使你能够在两个方向设置滚动单位的大小。接下来的两个参数noUnitsX和noUnitsY使你能够按滚动单位设置滚动区域的尺寸。换句话说，滚动区域的象素尺寸是(pixelsPerUnitX *noUnitsX*, *pixelsPerUnitY* noUnitsY)。例8.7通过将滚动单位设为1像素而避免了可能造成的混淆。参数xPos和yPos以滚动单位（非像素）为单位，它设置滚动条的初始位置，如果参数noRefresh为true，那么就阻止了在因SetScrollbars()的调用而引起的滚动后的窗口的自动刷新。

还有另外的三个方法，你可以用来设置滚动区域的尺寸，然后单独设置滚动率。你可能发现这些方法更容易使用，因为它们使你能够更直接地指定尺寸。你可以如下以像素为单位使用滚动窗口的SetVirtualSize()方法来直接设置尺寸。

```
self.scroll.SetVirtualSize((600, 400))
```

使用方法FitInside()，你可以在滚动区域中设置窗口部件，以便滚动窗口绑定它们。这个方法设置滚动窗口的边界，以使滚动窗口刚好适合其中的所有子窗口：

```
self.scroll.FitInside()
```

通常使用FitInside()的情况是，当在滚动窗口中正好有一个窗口部件（如文本域），并且该窗口部件的逻辑尺寸已被设置。如果我们在例8.7中使用了FitInside()，那么一个更小的滚动区域将被创建，因为该区域将正好匹配右下按钮的边缘，而没有多余的内边距。

最后，如果滚动窗口中有一个sizer设置，那么使用SetSizer()设置滚动区域为sizer所管理的窗口部件的尺寸。这是在一个复杂的布局中最常用的机制。关于sizer的更多细节参见第11章。

对于上述所有三种机制，滚动率需要去使用SetScrollRate()方法单独设置，如下所示：

```
self.scroll.SetScrollRate(1, 1)
```

参数分别是x和y方向的滚动单位尺寸。大于0的尺寸都是有效的。

滚动条事件

在例8.7中的按钮事件处理器，使用Scroll()方法程序化地改变滚动条的位置。这个方法需要滚动窗口的x和y坐标，使用的是滚动单位。

在第7章中，我们答应了你可以捕获的来自滚动条的事件列表，因为它们也被用来去控制滑块。表8.7列出了所有被滚动窗口内在处理的滚动事件。通常，许多你不会用到，除非你建造自定义窗口部件。

表**8.7** 滚动条的事件

EVT_SCROLL	当任何滚动事件被触发时发生。
EVT_SCROLL_BOTTOM	当用户移动滚动条到它的范围的最末端时触发（底边或右边，依赖于方向）。
EVT_SCROLL_ENDSCROLL	在微软的Windows中，任何滚动会话的结束都将触发该事件，不管是因鼠标拖动或按键按下。
EVT_SCROLL_LINEDOWN	当用户向下滚动一行时触发。
EVT_SCROLL_LINEUP	当用户向上滚动一行时触发。
EVT_SCROLL_PAGEDOWN	当用户向下滚动一页时触发。
EVT_SCROLL_PAGEUP	当用户向上滚动一页时触发。
EVT_SCROLL_THUMBRELEASE	用户使用鼠标拖动滚动条滚动不超过一页的范围，并释放鼠标后，触发该事件。
EVT_SCROLL_THUMBTRACK	滚动条在一页内被拖动时不断的触发。
EVT_SCROLL_TOP	当用户移动滚动条到它的范围的最始端时触发，可能是顶端或左边，依赖于方向而定。

行和页的准确定义依赖于你所设定的滚动单位，一行是一个滚动单位，一页是滚动窗口中可见部分的全部滚动单位的数量。对于表中所列出的每个EVT_SCROLL事件，都有一个相应的*EVT_SCROLLWIN*事件（它们由wx.ScrolledWindow产生）来回应。

还有一个wxPython的特殊的滚动窗口子类：wx.lib.scrolledpanel.ScrolledPanel，它使得你能够在面板上自动地设置滚动，该面板使用一个sizer来管理子窗口部件的布局。wx.lib.scrolledpanel.ScrolledPanel增加的好处是，它让用户能够使用tab键来在子窗口部件间切换。面板自动滚动，使新获得焦点的窗口部件进入视野。要使用wx.lib.scrolledpanel.ScrolledPanel，就要像一个滚动窗口一样声明它，然后，在所有的子窗口被添加后，调用下面的方法：

```
SetupScrolling(self, scroll_x=True, scroll_y=True, rate_x=20,
rate_y=20)
```

rate_x和rate_y是窗口的滚动单位，该类自动根据sizer所计算的子窗口部件的尺寸设定虚拟尺寸(virtual size)。 记住，当确定滚动窗口中的窗口部件的位置的时候，该位置总是窗口部件的物理位置，它相对于显示器中的滚动窗口的实际原点，而非窗口部件相对于显示器虚拟尺寸(virtual size)的逻辑位置。这始终是成立的，即使窗口部件不再可见。例如，在敲击了图8.5中的Scroll Me按钮后，该按钮所报告的它的位置是(-277,-237)。如果这不的你所想要的，那么使用CalcScrolledPosition(x,y)和。CalcUnscrolledPosition(x, y)方法在显示器坐标和逻辑坐标之间切换。在这两种情况中，在按钮敲击并使滚动条移动到底部后，你传递指针的坐标，并且滚动窗口返回一个(x,y)元组，如下所示：

```
CalcUnscrolledPostion(-277, -237) #
```

可选的框架类型

框架不限于其中带有窗口部件的普通的矩形，它可以呈现其它的形状。你也可以创建MDI（多文档界面）框架，它其中包含别的框架。或者你也可以去掉框架的标题栏，并且仍然可以使用户能拖动框架。

如何创建一个MDI框架？

还记得MDI吗？许多人都不记得了。MDI是微软90年代初的创新，它使得一个应用程序中的多个子窗口能被一个单一的父窗口控制，本质上为每个应用程序提供了一个独立的桌面。在大多数应用程序中，MDI要求应用程序中的所有窗口同时最小化，并保持相同的z轴次序（相对系统中的其它部分）。我们建议仅当用户期望同时看到所有的应用程序窗口的情况下使用MDI，例如一个游戏。图8.7显示了一个典型的MDI环境。

在wxPython中MDI是被支持的，在Windows操作系统下通过使用本地窗口部件来实现MDI，在其它的操作系统中通过模拟子窗口实现MDI。例8.8提供了一简单的MDI的例子。

图**8.7**

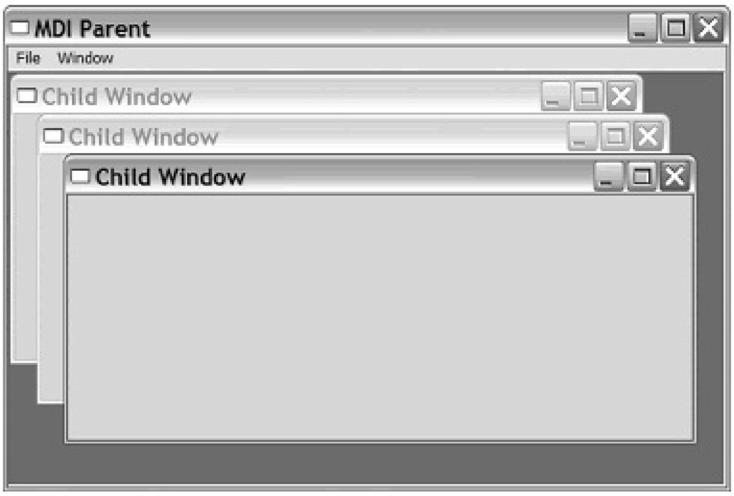


Figure 8.7
An MDI window

例**8.8** 如何创建一个**MDI**窗口

```
import wx

class MDIFrame(wx.MDIParentFrame):
    def __init__(self):
wx.MDIParentFrame.__init__(self, None, -1, "MDI Parent",
                size=(600,400))
        menu = wx.Menu()
        menu.Append(5000, " Window")
        menu.Append(5001, "E ")
        menubar = wx.MenuBar()
        menubar.Append(menu, "  ")
        self.SetMenuBar(menubar)
        self.Bind(wx.EVT_MENU, self.OnNewWindow, id=5000)
        self.Bind(wx.EVT_MENU, self.OnExit, id=5001)

    def OnExit(self, evt):
        self.Close(True)

    def OnNewWindow(self, evt):
        win = wx.MDIChildFrame(self, -1, "Child Window")
        win.Show(True)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MDIFrame()
    frame.Show()
    app.MainLoop()
```

MDI的基本概念是十分简单的。父窗口是wx.MDIParentFrame的一个子类，子窗口如同任何其它的wxPython窗口部件一样被添加，除了它们是wx.MDIChildFrame的子类。wx.MDIParentFrame的构造函数与wx.Frame的基本相同，如下所示：

```
wx.MDIParentFrame(parent, id, title, pos = wx.DefaultPosition,
size=wxDefaultSize,
style=wx.DEFAULT_FRAME_STYLE | wx.VSCROLL | wx.HSCROLL,
        name="frame")
```

不同的一点是wx.MDIParentFrame在默认情况下有滚动条。wx.MDIChildFrame的构造函数是相同的，除了它没有滚动条。如例8.8所示，添加一个子框架是通过创建一个以父框架为父亲的框架来实现的。

你可以通过使用父框架的Cascade()或Tile()方法来同时改变所有子框架的位置和尺寸，它们模拟相同名字的菜单项。调用Cascade()，导致一个窗口显示在其它的上面，如图8.7的所示，而Tile()使每个窗口有相同的尺寸并移动它们以使它们不重叠。要以编程的方式在子窗口中移动焦点，要使用父亲的方法ActivateNext()和ActivatePrevious()

什么是小型框架，我们为何要用它？

小型框架是一个有两个例外的矩形框架：它有一个较小的标题区域，并且在微软的Windows下或GTK下，它不在任务栏中显示。图8.8显示了一个较小标题域的一个例子。

图**8.8** 一个小型框架



Figure 8.8 A mini-frame in action

创建小型框架的代码基本上等同于创建一个矩形框架，唯一的不同是父类是wx.MiniFrame。例8.9显示了这个代码。

例**8.9** 创建一个小型框架

```
import wx

class MiniFrame(wx.MiniFrame):
    def __init__(self):
wx.MiniFrame.__init__(self, None, -1, 'Mini Frame',
                size=(300, 100))
        panel = wx.Panel(self, -1, size=(300, 100))
        button = wx.Button(panel, -1, "Close Me", pos=(15, 15))
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

    def OnCloseMe(self, event):
        self.Close(True)

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    MiniFrame().Show()
app.MainLoop()
```

wx.MiniFrame的构造函数等同于wx.Frame的，然而，wx.MiniFrame支持额外的样式标记。如表8.8所示。

表**8.8 wx.MiniFrame**的样式标记

wx.THICK_FRAME	在Windows或Motif下，使用粗边框绘制框架。
wx.TINY_CAPTION_HORIZONTAL	代替wx.CAPTION而显示一个较小的水平标题。
wx.TINY_CAPTION_VERTICAL	代替wx.CAPTION而显示一个较小的垂直标题。

典型的，小型框架被用于工具框窗口中，在工具框窗口中始终是有效的，它们不影响任务栏。较小的标题使得它们更有效的利用空间，并且明显地区别于标准的框架。

如何创建一个非矩形的框架？

在大多数应用程序中，框架都是矩形，因为矩形有一个不错的规则的形状，并且绘制和维护相对简单。可是，有时候你需要打破直线的制约。在wxPython中，你可以给框架一个任一的形状。如果一个备用的形状被定义了，那么框架超出该形状的部分不将被绘制，并且不响应鼠标事件；对于用户而言，它们不是框架的一部分。图8.9显示了一个非矩形的窗口，显示的背景是文本编辑器中的代码。

事件被设置来以便双击时开关这个非标准形状，鼠标右键单击时关闭这个窗口。这个例子使用了来自wxPython demo的images模块作为vippi的图像的资源，wxPython的吉祥物。

图**8.9**

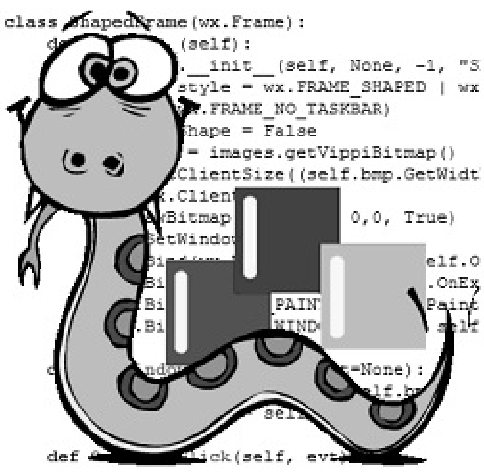


Figure 8.9
A window shaped into a familiar non-rectangular shape

例8.10显示了在这个非矩形框架后面的代码。这个例子比我们见过的其它一些稍微精细点，以显示如何在缺少典型的窗口界面装饰的情况下管理像窗口关闭之类的事情。

例**8.10** 绘制符合形状的窗口

```
import wx
import images

class ShapedFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Shaped Window",
                style = wx.FRAME_SHAPED | wx.SIMPLE_BORDER |
                wx.FRAME_NO_TASKBAR)
        self.hasShape = False

#1 获取图像
        self.bmp = images.getVippiBitmap()
        self.SetClientSize((self.bmp.GetWidth(), self.bmp.GetHei
ght()))

#2 绘制图像
        dc = wx.ClientDC(self)
        dc.DrawBitmap(self.bmp, 0,0, True)

        self.SetWindowShape()
        self.Bind(wx.EVT_LEFT_DCLICK, self.OnDoubleClick)
        self.Bind(wx.EVT_RIGHT_UP, self.OnExit)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_WINDOW_Create, self.SetWindowShape)#3
绑定窗口创建事件

    def SetWindowShape(self, evt=None):#4 设置形状
        r = wx.RegionFromBitmap(self.bmp)
        self.hasShape = self.SetShape(r)

    def OnDoubleClick(self, evt):
        if self.hasShape:
            self.SetShape(wx.Region())#5 重置形状
            self.hasShape = False
        else:
            self.SetWindowShape()

    def OnPaint(self, evt):
        dc = wx.PaintDC(self)
        dc.DrawBitmap(self.bmp, 0,0, True)

    def OnExit(self, evt):
        self.Close()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ShapedFrame().Show()
    app.MainLoop()
```

#1 在从images模块得到图像后，我们将窗口内部的尺寸设置为位图的尺寸。你也可以根据一个标准的图像文件来创建这个wxPython位图，这将在第16章中作更详细的讨论。

#2 这里，我们在窗口中绘制这个图像。这决不是一个必然的选择。你可以像其它窗口一样在该形状窗口中放置窗口部件和文本（尽管它们必须在该形状的区域内）。

#3 这个事件在大多数平台上是多余的，它强制性地在窗口被创建后调用SetWindowShape()。但是，GTK的实现要求在该形状被设置以前，窗口的本地UI对象被创建和确定，因此当窗口创建发生时我们使用窗口创建事件去通知并在它的处理器中设置形状。

#4 我们使用全局方法wx.RegionFromBitmap去创建设置形状所需的wx.Region对象。这是创建不规则形状的最容易的方法。你也可以根据一个定义多边形的点的列表来创建一个wx.Region。图像的透明部分的用途是定义区域的边界。

#5 双击事件开关窗口的形状。要回到标准的矩形，要使用一个空的wx.Region作为参数来调用SetShape()。

除了没有标准的关闭框或标题栏等外，不规则形状框架的行为像一个普通的框架一样。任何框架都可以改变它的形状，因为SetShape()方法是wx.Frame类的一部分，它可以被任何子类继承。在wx.SplashScreen中，符合形状的框架是特别的有用。

如何拖动一个没有标题栏的框架？

前一个例子的明显结果是这个没有标题栏的框架不能被拖动的，这儿没有拖动窗口的标准方法。要解决这个问题，我们需要去添加事件处理器来在当拖动发生时移动该窗口。例8.11显示与前一例子相同形状的窗口，但增加了对于处理鼠标左键敲击和鼠标移动的一些事件。这个技术可以适用于任何其它的框架，甚至是框架内你想要移动的窗口（例如绘画程序中的元素）。

例**8.11** 使用户能够从框架来拖动框架的事件

```
import wx
import images

class ShapedFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Shaped Window",
                style = wx.FRAME_SHAPED | wx.SIMPLE_BORDER )
        self.hasShape = False
        self.delta = wx.Point(0,0)
        self.bmp = images.getVippiBitmap()
        self.SetClientSize((self.bmp.GetWidth(), self.bmp.GetHei
ght()))
        dc = wx.ClientDC(self)
        dc.DrawBitmap(self.bmp, 0,0, True)
        self.SetWindowShape()
        self.Bind(wx.EVT_LEFT_DCLICK, self.OnDoubleClick)
```

```
#1 新事件
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)
        self.Bind(wx.EVT_LEFT_UP, self.OnLeftUp)
        self.Bind(wx.EVT_MOTION, self.OnMouseMove)

        self.Bind(wx.EVT_RIGHT_UP, self.OnExit)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_WINDOW_Create, self.SetWindowShape)

    def SetWindowShape(self, evt=None):
        r = wx.RegionFromBitmap(self.bmp)
        self.hasShape = self.SetShape(r)

    def OnDoubleClick(self, evt):
        if self.hasShape:
            self.SetShape(wx.Region())
            self.hasShape = False
        else:
            self.SetWindowShape()

    def OnPaint(self, evt):
        dc = wx.PaintDC(self)
        dc.DrawBitmap(self.bmp, 0,0, True)

    def OnExit(self, evt):
        self.Close()

    def OnLeftDown(self, evt):#2 鼠标按下
        self.CaptureMouse()
        pos = self.ClientToScreen(evt.GetPosition())
        oigin = self.GetPosition()
        self.delta = wx.Point(pos.x - oigin.x, pos.y - oigin.y)

    def OnMouseMove(self, evt):#3 鼠标移动
        if evt.Dragging() and evt.LeftIsDown():
            pos = self.ClientToScreen(evt.GetPosition())
            newPos = (pos.x - self.delta.x, pos.y - self.delta.y
)
            self.Move(newPos)

    def OnLeftUp(self, evt):#4 鼠标释放
        if self.HasCapture():
            self.ReleaseMouse()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ShapedFrame().Show()
    app.MainLoop()
```

#1 我们为三个事件增加了相应的处理器，以作相应的工作。这三个事件是鼠标左键按下，鼠标左键释放和鼠标移动。

#2 拖动事件从鼠标左键按下开始。这个事件处理器做两件事。首先它捕获这个鼠标，直到鼠标被释放，以防止鼠标事件被改善到其它窗口部件。第二，它计算事件发生的位置和窗口左上角之间的偏移量，这个偏移量将被用来计算窗口的新位置。

#3 这个处理器当鼠标移动时被调用，它首先检查看该事件是否是一个鼠标左键按下，如果是，它使用这个新的位置和前面计算的偏移量来确定窗口的新位置，并移动窗口。

#4 当鼠标左键被释放时，ReleaseMouse()被调用，这使得鼠标事件又可以被发送到其它的窗口部件。

这个拖动技术可以被完善以适合其它的需要。例如，仅在一个定义的区域内鼠标敲击才开始一个拖动，你可以对鼠标按下事件的位置做一个测试，使敲击发生在右边的位置时，才能拖动。

使用分割窗

分割窗是一种特殊的容器窗口部件，它管理两个子窗口。这两个子窗口可以被水平的堆放或彼此左右相邻。在两个子窗口之间的是一个窗框，它是一个可移动的边框，移动它就改变了两个子窗口的尺寸。分割窗经常被用于主窗口的侧边栏。图8.10显示了一个分割窗的样板。

当你有两个信息面板并且想让用户自主决定每个面板的尺寸时，可以使用分割窗。Mac OS X Finder窗口就是一个分割窗的例子，并且许多的文本编辑器或制图软件都用它来维护一个打开的文件的列表。

创建一个分割窗

在wxPython中，分割窗是类wx.SplitterWindow的实例。和大多数其它的wxPython窗口部件不一样，分隔窗口在被创建后，可使用前要求进一步的初始化。它的构造函数是十分简单的。

图**8.10**



Figure 8.10 A sample splitter window after initialization

```
wx.SplitterWindow(parent, id=-1, pos=wx.DefaultPosition,
size=wx.DefaultSize, style=wx.SP_3D,
name="splitterWindow")
```

它的这些参数都有标准的含义——parent是窗口部件的容器，pos是窗口部件在它的父容器中位置，size是它的尺寸。

在创建了这个分割窗后，在它可以被使用前，你必须对这个窗口调用三个方法中的一处。如果你想初始时只显示一个子窗口，那么调用Initialize(window)，参数window是这个单一的子窗口（通常是一种wx.Panel）。在这种情况下，窗口将在以后响应用户的动作时再分割。

要显示两个子窗口，使用SplitHorizontally (window1，window2，sashPosition=0)或SplitVertically(window1, window2, sashPosition=0)。两个方法的工作都是相似的，参数window1和window2包含两个子窗口，参数sashPosition包含分割条的初始位置。对于水平分割（水平分割条）来说，window1被放置在window2的顶部。如果sashPosition是一个正数，它代表顶部窗口的初始高度（也就是分割条距顶部的像素值）。如果sashPosition是一个负数，它定义了底部窗口的尺寸，或分割条距底部的像素值。如果sashPosition是0，那么这个分割条位于正中。对于垂直分割（垂直分割条），window1位于左边，window2位于右边。正值的sashPosition设置window1的尺寸，也就是分割条距左边框的像素值。类似的，负值sashPosition设置右边子窗口的尺寸，0值将分割条放置在正中。如果你的子窗口复杂的话，我们建议你在布局中使用sizer，以便于当分割条被移动时很好地调整窗口的大小。

一个分割窗的例子

例8.12显示了如何创建有一个子窗口的分割窗并且在以后响应菜单项的分割。这个例子也使用了一些事件，这些事件我们以后讨论。注意，我们不计划在开始可见的子面板使用Hide()方法来隐藏。我们这样做是因为我们开始时不告诉分割窗去管理

那个子面板的尺寸和位置，所以我们使用这种方法来隐藏它。如果我们在开始就要分割和显示这两个子面板，那么我们就不必考虑这些。

例**8.12** 如何创建你自己的分割窗

```
import wx

class SplitterExampleFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, title=title)
        self.MakeMenuBar()
        self.minpane = 0
        self.initpos = 0
        self.sp = wx.SplitterWindow(self)# 创建一个分割窗
        self.p1 = wx.Panel(self.sp, style=wx.SUNKEN_BORDER)# 创建
子面板
        self.p2 = wx.Panel(self.sp, style=wx.SUNKEN_BORDER)
        self.p1.SetBackgroundColour("pink")
        self.p2.SetBackgroundColour("sky blue")
        self.p1.Hide()# 确保备用的子面板被隐藏
        self.p2.Hide()

        self.sp.Initialize(self.p1)# 初始化分割窗

        self.Bind(wx.EVT_SPLITTER_SASH_POS_CHANGING,
                  self.OnSashChanging, self.sp)
        self.Bind(wx.EVT_SPLITTER_SASH_POS_CHANGED,
                  self.OnSashChanged, self.sp)

    def MakeMenuBar(self):
        menu = wx.Menu()
        item = menu.Append(-1, "Split horizontally")
        self.Bind(wx.EVT_MENU, self.OnSplitH, item)
        self.Bind(wx.EVT_Update_UI, self.OnCheckCanSplit, item)
        item = menu.Append(-1, "Split vertically")
        self.Bind(wx.EVT_MENU, self.OnSplitV, item)
        self.Bind(wx.EVT_Update_UI, self.OnCheckCanSplit, item)
        item = menu.Append(-1, "Unsplit")
        self.Bind(wx.EVT_MENU, self.OnUnsplit, item)
        self.Bind(wx.EVT_Update_UI, self.OnCheckCanUnsplit, item
)

        menu.AppendSeparator()
        item = menu.Append(-1, "Set initial sash position")
        self.Bind(wx.EVT_MENU, self.OnSetPos, item)
        item = menu.Append(-1, "Set minimum pane size")
        self.Bind(wx.EVT_MENU, self.OnSetMin, item)

        menu.AppendSeparator()
        item = menu.Append(wx.ID_EXIT, "E ")
        self.Bind(wx.EVT_MENU, self.OnExit, item)
```

```
        mbar = wx.MenuBar()
        mbar.Append(menu, "Splitter")
        self.SetMenuBar(mbar)

    def OnSashChanging(self, evt):
        print "OnSashChanging:", evt.GetSashPosition()

    def OnSashChanged(self, evt):
        print "OnSashChanged:", evt.GetSashPosition()

    def OnSplitH(self, evt):# 响应水平分割请求
        self.sp.SplitHorizontally(self.p1, self.p2, self.initpos
)

    def OnSplitV(self, evt):# 响应垂直分割请求
        self.sp.SplitVertically(self.p1, self.p2, self.initpos)

    def OnCheckCanSplit(self, evt):
        evt.Enable(not self.sp.IsSplit())

    def OnCheckCanUnsplit(self, evt):
        evt.Enable(self.sp.IsSplit())

    def OnUnsplit(self, evt):
        self.sp.Unsplit()

    def OnSetMin(self, evt):
        minpane = wx.GetNumberFromUser(
            "Enter the minimum pane size",
            "", "Minimum Pane Size", self.minpane,
            0, 1000, self)
        if minpane != -1:
            self.minpane = minpane
            self.sp.SetMinimumPaneSize(self.minpane)

    def OnSetPos(self, evt):
        initpos = wx.GetNumberFromUser(
            "Enter the initial sash position (to be used in the
Split call)",
            "", "Initial Sash Position", self.initpos,
            -1000, 1000, self)
        if initpos != -1:
            self.initpos = initpos

    def OnExit(self, evt):
        self.Close()

app = wx.PySimpleApp(redirect=True)
frm = SplitterExampleFrame(None, "Splitter Example")
frm.SetSize((600,500))
frm.Show()
app.SetTopWindow(frm)
app.MainLoop()
```

分割窗只能分割一次，对已分割的窗口再分割将会失败，从而导致分割方法返回False（成功时返回True）。要确定窗口当前是否被分割了，调用方法IsSplit()。在例8.12中，为了确保相应的菜单项有效，就采用这个方法。

如果你想不分割窗口，那么使用Unsplit(toRemove=None)。参数toRemove是实际要移除的wx.Window对象，并且必须是这两个子窗口中的一个。如果toRemove是None，那么底部或右部的窗口将被移除，这根据分割的方向而定。默认情况下，被移除的窗口是没有被wxPython删除的，所以以后你可以再把它添加回来。unsplit方法在取消分割成功时返回True。如果分割窗当前没有被分割，或toRemove参数不是两个子窗口中的一个，那么该方法返回False。

要确保你对想要的子窗口有一个正确的引用，那么使用GetWindow1()和GetWindow2()方法。GetWindow1()方法返回顶部或左边的子窗口，而GetWindow2()方法返回底部或右边的窗口。由于没有一个直接的设置方法来改变一个子窗口，所以使用方法ReplaceWindow(winOld, winNew)，winOld是你要替换的wx.Window对象，winNew是要显示的新窗口。

改变分割的外观

有许多样式标记用来控制显示在屏幕上的分割窗的外观。注意，由于分割与平台有关，所以不是所有列出的标记都将对任何平台起作用。表8.9说明了这些有效的标记。

我们将在接下来的部分看到，你也可以用你的程序来改变分割的显示，以响应用户的动作或你自己的要求。

表**8.9** 分割窗的样式

wx.SP_3D	绘制三维的边框和分割条。这是一个默认样式。
wx.SP_3DBORDER	绘制三维样式的边框，不包括分割条。
wx.SP_3DSASH	绘制三维样式的分割条，不包括边框。
wx.SP_BORDER	绘制窗口的边框，非三维的样式。
wx.SP_LIVE_Update	改变响应分割条移动的默认行为。如果没有设置这个标记，那么当用户拖动分割条时，将绘制一条线来标明分割条的新位置。子窗口的尺寸没有被实际地更新，直到完成分割条拖放。如果设置了这个标记，那么当分割条在被拖动时，子窗口的尺寸将不断地变化。
wx.SP_NOBORDER	不绘制任何边框。
wx.SP_NO_XP_THEME	在Windows XP系统下，分割条不使用XP的主题样式，它给窗口一个更经典的外观。
wx.SP_PERMIT_UNSPLIT	如果设置了这个样式，那么窗口始终不被分割。如果不设置，你可以通过设置大于0的最小化的窗格尺寸来防止窗口被分割。

以程序的方式处理分割

一旦分割窗被创建，你就可以使用窗口的方法来处理分割条的位置。特别是，你可以使用方法SetSashPosition(position,redraw=True)来移动分割条。position是以像素单位的新的位置，它是分割条距窗口顶部或左边的距离。用在分割方法中的负值，表示位置从底部或右边算起。如果redraw为True，则窗口立即更新。否则它等待常规窗口的刷新。如果你的像素值在范围外的话，设置方法的行为将不被定义。要得到当前分割条的位置，使用GetSashPosition()方法。

在默认的分割行为下，用户可以在两个边框间随意移到分割条。移动分割条到一边，使得别一子窗口的尺寸为0，这导致窗口此时成未分割状态。要防止这种情况，你可以使用方法SetMinimumPaneSize(paneSize)来指定子窗口的最小尺寸。paneSize参数是子窗口的最小像素尺寸。这样，用户就不能通过拖放来使子窗口更小，程序同样也不能使子窗口更小。如前所述，你可以使用wx.SP_PERMIT_UNSPLIT样式来达到相同的效果。要得到当前最小子窗口尺寸，使用方法GetMinimumPaneSize()。

改变窗口的分割模式，使用方法SetSplitMode(mode)，参数mode取下列常量值之一：wx.SPLIT_VERTICAL、wx.SPLIT_HORIZONTAL。如果模式改变了，那么顶部窗口变为左边，而底部变为右边（反之亦然）。该方法不引起窗口的重绘，你必须显式地进行强制重绘。你可以使用GetSplitMode()来得到当前的分割模式，它返回上面两个常量值之一。如果窗口当前是未分割状态，那么GetSplitMode()方法返回最近的分割模式。

典型的，如果wx.SP_LIVE_Update样式没有被设置，那么子窗口仅在分割条拖动会话结束时改变尺寸。如果你想在其它时刻强制子窗口重绘，你可以使用方法UpdateSize()。

响应分割事件

分割窗触发wx.SplitterEvent类事件。这儿有四个不同的分割窗的事件类型，如表8.10所示。

表**8.10** 分割窗的事件类型

EVT_SPLITTER_DCLICK	当分割条被双击时触发。捕捉这个事件不阻塞标准的不分割行为，除非你调用事件的Veto()方法。
EVT_SPLITTER_SASH_POS_CHANGED	分割条的改变结束后触发，但在此之前，改变将在屏幕上显示（因此你可以再作用于它）。这个事件可以使用Veto()来中断。
EVT_SPLITTER_SASH_POS_CHANGING	当分割条在被拖动时，不断触发该事件。这个事件可以通过使用事件的Veto()方法来中断，如果被中断，那么分割条的位置不被改变。
EVT_SPLITTER_UNSPLIT	变成未分割状态时触发。

这个分割事件类是wx.CommandEvent的子类。从分割事件的实例，你可以访问关于分割窗当前状态的信息。对于涉及到分割条移动的两个事件，调用GetSashPosition()得到分割条相对于左或顶部的位置，这依据分割条的方向而定。在位置正在变化事件中，调用SetSashPosition(pos)，将用线条表示新的位置。在位置已改变事件中，SetSashPosition(pos)方法将移动分割条。对于双击事件，你可以使用事件的GetX() 和GetY()方法得到敲击的确切位置。对于未分割事件，你可以使用GetWindowBeingRemoved()方法来得到哪个窗口被移除了。

本章小结

1、wxPython中的大部分用户交互都发生在wx.Frame或wx.Dialog中。wx.Frame代表用户调用的窗口。wx.Frame实例的创建就像其它的wxPython窗口部件一样。wx.Frame的典型用法包括创建子类，子类通过定义子窗口部件，布局和行为来扩展基类。通常，一个框架包含只包含一个wx.Panel的顶级子窗口部件或别的容器窗口。

2、这儿还有各种特定于wx.Frame的样式标记。其中的一些影响框架的尺寸和形状，另一些影响在系统中相对于其它的框架，它将如何被绘制，还有一些定义了在框架边框上有那些界面装饰。在某种情况下，定义一个样式标记需要“两步”的创建过程。

3、通过调用Close()方法可以产生关闭框架的请求。这给了框架一个关闭它所占用的资源的机会。框架也能否决一个关闭请求。调用Destroy()方法将迫使框架立即消失而没有任何延缓。

4、框架中的一个特定的子窗口部件可以使用它的wxPython ID、名字或它的文本标签来发现。

5、通过包括wx.ScrolledWindow类的容器部件可以实现滚动。这儿有几个方法来设置滚动参数，最简单的是在滚动窗口中使用sizer，在这种情况下，wxPython自动确定滚动面板的虚拟尺寸（virtual size）。如果想的话，虚拟尺寸可以被手动设置。

6、这儿有一对不同的框架子类，它们允许不同的外观。类wx.MDIParentFrame可以被用来创建MDI，而wx.MiniFrame可以创建一个带有较小标题栏的工具框样式的窗口。使用SetShape()方法，框架可以呈现出非矩形的形状。形状的区域可以被任何位图定义，并使用简单的颜色掩码来决定区域的边缘。非矩形窗口通常没有标准的标题栏，标题栏使得框架可以被拖动， 但这可以通过显式地处理鼠标事件来管理。

7、位于两个子窗口间的可拖动的分割条能够使用wx.SplitterWindow来实现，分割条可以被用户以交互的方式处理，或以编程的方式处理（如果需要的话）。

在下一章，我们将讨论对话框，它的行为类似于框架。

第九章 通过对话框让用户选择

1. 使用模式对话框工作
2. 使用标准对话框
3. 创建向导
4. 显示启动提示
5. 使用验证器（validator）来管理对话框中的数据
6. 本章小结

模式对话框用于与用户进行快速的交互或在用户可以执行程序的下一步之前，对话框中的信息必须被输入的时候。在 `wxPython` 中，有几个标准的函数用来显示基本的模式对话框。这些对话框包括警告框，单行文本域，和从列表中选择。在随后的部分，我们将给你展示这些对话框，以及如何使用这些预定义的函数来减轻你的工作量。

如何创建一个模式对话框？

模式对话框阻塞了别的窗口部件接收用户事件，直到该模式对话框被关闭；换句话说，在它存在期间，用户一直被置于对话模式中。如图9.1所示，你不能总是根据外观来区别对话框和框架。在 `wxPython` 中，对话框与框架间的区别不是基于它们的外观的，而主要是它们处理事件的办法的实质。

图**9.1** 一个模式对话框



Figure 9.1
A sample modal dialog

对话框的创建和配置与框架稍微有些不同。例9.1显示了产生图9.1的代码。所显示的对话框上的按钮被敲击后，该对话框就关闭了，并且一条消息被输出到 `stdout(` 标准输出)。

例**9.1** 定义一个模式对话框

```
import wx

class SubclassDialog(wx.Dialog):
    def __init__(self):#初始化对话框
wx.Dialog.__init__(self, None, -1, 'Dialog Subclass',
                size=(300, 100))
        okButton = wx.Button(self, wx.ID_OK, "OK", pos=(15, 15))
        okButton.SetDefault()
cancelButton = wx.Button(self, wx.ID_CANCEL, "Cancel",
                pos=(115, 15))

if __name__ == '__main__':
    app = wx.PySimpleApp()
app.MainLoop()
    dialog = SubclassDialog()
    result = dialog.ShowModal()#显示模式对话框
    if result == wx.ID_OK:
        print "OK"
    else:
        print "Cancel"
    dialog.Destroy()
```

与前一章的 `wx.Frame` 的例子比较，这儿有两个需要注意的事情。
在 `__init__` 方法中，按钮是被直接添加到 `wx.Dialog` ，而非 `wx.Panel` 。面板在对话框中的使用比在框架中少的多，部分原因是因为对话框与框架相比倾向简单化，但主要是因为 `wx.Panel` 特性（标准系统背景和 `tab` 键横向切换控件焦点）已经默认存在于 `wx.Dialog` 中。

要显示为模式对话框，使用 `ShowModal()` 方法。这与用于框架的的 `Show()` 方法在对程序的执行上有不同的作用。在调用 `ShowModal()` 后你的应用程序将处于等待中，直到对话框被取消。

模式将保持到对话框方法 `EndModal(retCode)` 被调用，该方法关闭对话框。参数 `retCode` 是由 `ShowModal()` 方法返回的一个整数值。典型的，应用程序利用这个返回值来知道用户是如何关闭对话框的，以控制以后的操作。但是结束这个模式并没有销毁或甚至关闭对话框。保持对话框的存在可能是一件好事，因为这意味你可以把用户选择的信息存储为对话框实例的数据成员，并且即使在对话框被关闭后也能从对话框重新获得那些信息。在接下来的部分，我们将看一些我们使用对话框处理程序中用户输入的数据的例子。

由于例9.1中没有定义事件处理器，你可能会惊奇对话框是如何响应按钮敲击的。这个行为已经定义在 `wxDialog` 中了。有两个预定义的 `wxPython` `ID` 号，它们在对话框中有特殊的意思。当对话框中的一个使用 `wx.ID_OK` `ID` 的 `wx.Button` 被敲击时，模式就结束了，对话框也关闭了， `wx.ID_OK` 就是 `ShowModal()` 调用返回的值。同样，一个使用 `wx.ID_CANCEL` `ID` 的按钮做相同的事情，但是 `ShowModal()` 的返回值是 `wx.ID_CANCEL` 。

例9.1显示了处理模式对话框的一个典型的方法。在对话框被调用后，返回值被用作 `if` 语句中的测试。在这种情况下，我们简单地打印结果，在更复杂的例子中， `wx.ID_OK` 将执行用户在对话框中所要求的动作，如打开文件或选择颜色。

典型的，你在完成对对话框的使用后，你应该显式地销毁它。这通知C++对象它应该自我销毁，然后这将使得它的 `Python` 部分被作为垃圾回收。如果你希望在你的应用程序中，以后再次使用该对话框时不重建它，以加速对话框的响应时间，那么你可以保持对该对话框的一个引用，并当你需要再次激活它时，简单地调用它的 `ShowModal()` 方法。当应用程序准备退出时，确保已销毁了它，否则 `MainLoop()` 将仍将它作为一个存在的顶级窗口，并且程序将不能正常退出。

如何创建一个警告框？

经由一个对话框与用户交互的最简单的三个办法分别是： `wx.MessageDialog` ，它是一个警告框、 `wx.TextEntryDialog` ，它提示用户去输入一些短的文本、 `wx.SingleChoiceDialog` ，它使用户能够从一个有效选项列表中进行选择。在接下来的三个小节中，我们将论这些简单的对话框。

消息对话框显示一个短的消息，并使用户通过按下按钮来作响应。通常，消息框被用作去显示重要的警告、 `yes` / `no` 问题、或询问用户是否继续某种操作。图9.2显示了一个典型的消息框。

图**9.2**



Figure 9.2 A standard message box, in a yes/no configuration

使用消息框是十分的简单。例9.2显示了创建一个消息框的两种办法。

例**9.2** 创建一个消息框

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()

# 方法一，使用类
    dlg = wx.MessageDialog(None, "Is this explanation OK?",
                           'A Message Box',
                           wx.YES_NO | wx.ICON_QUESTION)
    retCode = dlg.ShowModal()
    if (retCode == wx.ID_YES):
        print "yes"
    else:
        print "no"
    dlg.Destroy()

# 方法二，使用函数
    retCode = wx.MessageBox("Is this way easier?", "Via Function
",
        wx.YES_NO | wx.ICON_QUESTION)
```

例9.2创建了两个消息框，一个在另一个的后面。这第一个方法是创建类 `wx.MessageDialog` 的一个实例，并使用 `ShowModal()` 来显示它。

使用 **`wx.MessageDialog`** 类

使用 `wx.MessageDialog` 的构造函数，你可以设置对话框的消息和按钮，构造函数如下：

`wx.MessageDialog(parent` , `message` , `caption` =" `Message` `box` ",

- `style` = `wx.OK` | `wx.CANCEL` , `pos` = `wx.DefaultPosition)`

`message` 参数是实际显示在对话框中的文本。如果消息字符串包含\n字符，那么文本将在此换行。 `caption` 参数显示在对话框的标题栏中。 `pos` 参数使你可以指定对话框显示在屏幕上的位置——在微软 `Windows` 下，这个参数将被忽略。

`wx.MessageDialog` 的样式标记分为两类。第一类控制显示在对话框中的按钮。表9.1说明了这些样式。

表9.1 **`wx.MessageDialog`** 的按钮样式

`wx.CANCEL`	包括一个 `cancel` （取消）按钮。这个按钮有一个 `ID` 值 `wx.ID_CANCEL` 。
`wx.NO_DEFAULT`	在一个 `wx.YES_NO` 对话框中， `No` （否）按钮是默认的。
`wx.OK`	包括一个 `OK` 按钮，这个按钮有一个 `ID` 值 `wx.ID_OK` 。
`wx.YES_DEFAULT`	在一个 `wx.YES_NO` 对话框中， `Yes` 按钮是默认的。这是默认行为。
`wx.YES_NO`	包括 `Yes` 和 `No` 按钮，各自的 `ID` 值分别是 `wx.ID_YES` 和 `wx.ID_NO` 。

第二套样式标记控制紧挨着消息文本的图标。它们显示在表9.2中。

表**9.2** **`wx.MessageDialog`** 的图标样式

`wx.ICON_ERROR`	表示一个错误的图标。
`wx.ICON_EXCLAMATION`	表示警告的图标。
`wx.ICON_HAND`	同 `wx.ICON_ERROR` 。
`wx.ICON_INFORMATION`	信息图标，字母i。
`wx.ICON_QUESTION`	问号图标。

最后，你可以使用样式 `wx.STAY_ON_TOP` 将对话框显示在系统中任何其它窗口的上面，包括系统窗口和 `wxPython` 应用程序窗口。

你在例9.2所见到的，对话框通过使用 `ShowModal()` 被调用。根据所显示的按钮，返回的结果是以下值之一： `wx.ID_OK` , `wx.ID_CANCEL` ， `wx.ID_YES` , 或 `wx.ID_NO` 。如同其它对话框的情况，你通常使用这些值来控制程序的执行。

使用 **`wx.MessageBox()`** 函数

例9.2中的#1显示了一个调用消息框的更简短的方法。这个便利的函数 `wx.MessageBox()` 创建对话框，调用 `ShowModal()` ，并且返回下列值之一： `wx.YES` , `wx.NO` , `wx.CANCEL` , 或 `wx.OK` 。函数的形式比 `MessageDialog` 的构造函数更简单，如下所示：

```
wx.MessageBox(message, caption="Message", style=wx.OK)
```

在这个例子中，参数 `message` , `caption` , `style` 的意思和构造函数中的相同，你可以使用所有相同的样式标记。正如我们贯穿本章将看到的，在 `wxPython` 预定义的几个对话框都有便利的函数。在你为单一的使用创建对话框

的时候，你的选择有一个优先的问题。如果你计划束缚住对话框以便多次调用它，那么你可能会优先选择去实例化对象以便你能够束缚该引用，而不使用函数的方法，尽管这对于这些简单的对话框来说，所节约的时间可以忽略不计。

要在你的消息框中显示大量的文本（例如，终端用户许可证的显示），你可以使用 `wxPython` 特定的类 `wx.lib.dialogs.ScrolledMessageDialog` ，它包含如下的构造函数：

```
wx.lib.dialogs.ScrolledMessageDialog(parent, msg, caption,
        pos=wx.wxDefaultPosition, size=(500,300))
```

这个对话框不使用本地消息框控件，它根据别的 `wxPython` 窗口部件来创建一个对话框。它只显示一个 `OK` 按钮，并且没有更多的样式信息。

如何从用户得到短的文本？

这第二个简单类型的对话框是 `wx.TextEntryDialog` ，它被用于从用户那里得到短的文本输入。它通常用在在程序的开始时要求用户名或密码的时候，或作为一个数据输入表单的基本替代物。图9.3显示了一个典型的文本对话框。

图**9.3** 文本输入标准对话框

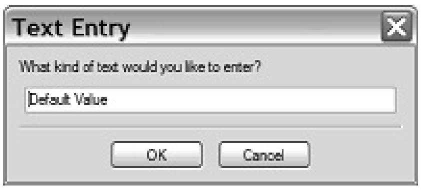


Figure 9.3
A text entry standard dialog

例9.3显示了产生图9.3的代码

例**9.3**

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
dialog = wx.TextEntryDialog(None,
            "What kind of text would you like to enter?",
            "Text Entry", "Default Value", style=wx.OK|wx.CANCEL
)
    if dialog.ShowModal() == wx.ID_OK:
        print "You entered: %s" % dialog.GetValue()

    dialog.Destroy()
```

在前一小节，我们创建了一个对话框类的实例，在这里，我们要用到的对话框类是`wx.TextEntryDialog`。该类的构造函数比简单消息对话框要复杂一些：

```
wx.TextEntryDialog(parent, message, caption="Please enter text",
defaultValue="", style=wx.OK | wx.CANCEL | wx.CENTRE,
    pos=wx.DefaultPosition)
```

`message`参数是显示在对话框中的文本提示，而`caption`显示在标题栏中。`defaultValue`显示在文本框中的默认值。`style`可以包括`wx.OK`和`wx.CANCEL`，它显示适当的按钮。

几个`wx.TextCtrl`的样式也可以用在这里。最有用的应该是`wx.TE_PASSWORD`，它掩饰所输入的真实密码。你也可以使用`wx.TE_MULTILINE`来使用户能够在对话框中输入多行文本，也可以使用`wx.TE_LEFT`，`wx.TE_CENTRE`，和`wx.TE_RIGHT`来调整所输入的文本的对齐位置。

例9.3的最后显示了在文本框和对话框之间的另一区别。用户所输入的信息被存储在该对话框实例中，并且以后必须应用程序获取。在这种情况下，你可以使用对话框的`GetValue()`方法来得到该值。记住，如果用户按下`Cancel`（取消）去退出该对话框，这意味他们不想去使用他所键入的值。你也可以在程序中使用`SetValue()`方法来设置该值。

下面这些是使用文本对话框的便利函数：

1、`wx.GetTextFromUser()` 2、`wx.GetPasswordFromUser()`
3、`wx.GetNumberFromUser()`

其中和例9.3的用处最近似的是`wx.GetTextFromUser()`：

```
wx.GetTextFromUser(message, caption="Input text",
    default_value="", parent=None)
```

这里的`message`，`caption`，`default_value`，和`parent`与`wx.TextEntryDialog`的构造函数中的一样。如果用户按下`OK`，该函数的返回值是用户所输入的字符串。如果用户按下`Cancel`，该函数返回空字符串。

如果你希望用户输入密码，你可以使用`wx.GetPasswordFromUser()`函数：

```
wx.GetPasswordFromUser(message, caption="Input text",
    default_value="", parent=None)
```

这里的参数意义和前面的一样。用户的输入被显示为星号，如果用户按下`OK`，该函数的返回值是用户所输入的字符串。如果用户按下`Cancel`，该函数返回空字符串。

最后，你可以使用 `wx.GetNumberFromUser()` 要求用户输入一个数字：

```
wx.GetNumberFromUser(message, prompt, caption, value, min=0,
max=100, parent=None)
```

这里的参数的意义有一点不同， `message` 是显在 `prompt` 上部的任意长度的消息， `value` 参数是默认显示在文本框中的长整型值。 `min` 和 `max` 参数为用户的输入限定一个范围。如果用户按下 `OK` 按钮退出的话，该方法返回所输入的值，并转换为长整型。如果这个值不能转换为一个数字，或不在指定的范围内，那么该函数返回-1，这意味如果你将该函数用于负数的范围的话，你可能要考虑一个转换的方法。

如何用对话框显示选项列表？

如果给你的用户一个空的文本输入域显得太自由了，那么你可以使用 `wx.SingleChoiceDialog` 来让他们在一组选项中作单一的选择。图9.4显示了一个例子。

图**9.4** 一个单选对话框



Figure 9.4 A single choice dialog

例9.4显示了产生图9.4的代码

例**9.4** 显示一个选择列表对话框

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    choices = ["Alpha", "Baker", "Charlie", "Delta"]
    dialog = wx.SingleChoiceDialog(None, "Pick A Word", "Choices
",
            choices)
    if dialog.ShowModal() == wx.ID_OK:
        print "You selected: %s\n" % dialog.GetStringSelection()

    dialog.Destroy()
```

`wx.SingleChoiceDialog` 的构造函数如下所示：

```
wx.SingleChoiceDialog(parent, message, caption, choices,
clientData=None, style=wx.OK | wx.CANCEL | wx.CENTRE,
pos=wx.DefaultPosition)
```

`message` 和 `caption` 参数的意义与前面的一样，分别显示在对话框和标题栏中。 `choices` 参数要求一个字符串的列表，它们是你呈现在对话框中的选项。 `style` 参数有三个项，这是默认的，分别是 `OK` 按钮、 `Cancle` 按钮和使对话框在屏幕中居中。 `centre` 选项和 `pos` 参数在 `Windows` 操作系统上不工作。

如果你想在用户看见对话框之前，设置它的默认选项，使用 `SetSelection(selection)` 方法。参数 `selection` 是选项的索引值，而非实际选择的字符串。在用户选择了一个选项后，你即可以使用 `GetSelection()` ——它返回所选项的索引值，也可以使用 `GetStringSelection()` ——它返回实际所选的字符串，来得到它。

有两个用于单选对话框的便利函数。第一个是 `wx.GetSingleChoice` ，它返回用户所选的字符串：

`wx.GetSingleChoice(message` , `caption` , `aChoices` , `parent` = `None)`

参数 `message` , `caption` , 和 `parent` 的意义和 `wx.SingleChoiceDialog` 构造函数的一样。 `aChoices` 参数是选项的列表。如果用户按下 `OK` ，则返回值是所选的字符串，如果用户按下 `Cancel` ，则返回值是空字符串。这意味如果空字符是一个有效的选择的话，那么你就不该使用这个函数。

第二个是 `wx.GetSingleChoiceIndex:`

```
wx.GetSingleChoiceIndex(message, caption, aChoices, parent=None)
```

这个函数与第一个有相同的参数，但是返回值不同。如果用户按下 `OK` ，则返回值是所选项的索引，如果用户按下 `Cancel` ，则返回值是-1。

如何显示进度条？

在许多程序中，程序需要自己做些事情而不受用户输入的干扰。这时就需要给用户一些可见的显示，以表明程序正在做一些事情及完成的进度。在 `wxPython` 中，这通常使用一个进度条来管理，如图9.5所示。

图**9.5**

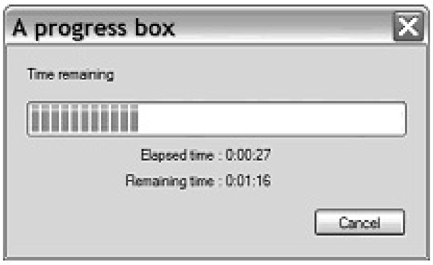


Figure 9.5 A sample progress box, joined in progress

例9.5显示了产生图9.5的代码

例**9.5** 生成一个进度条

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    progressMax = 100
    dialog = wx.ProgressDialog("A progress box", "Time remaining
", progressMax,
            style=wx.PD_CAN_ABORT | wx.PD_ELAPSED_TIME | wx.PD_R
EMAINING_TIME)
    keepGoing = True
    count = 0
    while keepGoing and count   progressMax:
        count = count + 1
        wx.Sleep(1)
        keepGoing = dialog.Update(count)

    dialog.Destroy()
```

进度条的所有选项在构造函数中被设置，构造函数如下：

```
wx.ProgressDialog(title, message, maximum=100, parent=None,
        style=wx.PD_AUTO_HIDE | wx.PD_APP_MODAL)
```

这些参数不同于其它对话框的。参数 `title` 被放置在窗口的标题栏， `message` 被显示在对话框中。 `maximum` 是你用来显示进度计数的最大值。

表9.3 列出了特定于 `wx.ProgressDialog` 六个样式，它们影响进度条的行为。

表**9.3** **`wx.ProgressDialog`** 的样式

wx.PD_APP_MODAL	如果设置了这个样式，进度条对整个应用程序是模式的，这将阻塞所有的用户事件。如果没有设置这个样式，那么进度条仅对它的父窗口是模式的。
wx.PD_AUTO_HIDE	进度条将自动隐藏自身直到它达到它的最大值。
wx.PD_CAN_ABORT	在进度条上放上一个 Cancel 按钮，以使用户停止。如何响应来自该对话框的取消将在以后说明。
wx.PD_ELAPSED_TIME	显示该对话框已经出现了多长时间。
wx.PD_ESTIMATED_TIME	显示根据已花的时间、当前的计数值和计数器的最大值所估计出的完成进度所需的总时间。
wx.PD_REMAINING_TIME	显示要完成进度所估计的剩余时间，或(所需总时间-已花时间)。

要使用进度条，就要调用它的唯一的方法 `Update(value` , `newmsg` ="")。 `value` 参数是进度条的新的内部的值，调用 `update` 将导致进度条根据新的计数值与最大计算值的比例重绘。如果使用可选的参数 `newmsg` ，那么进度条上的文本消息将变为该字符串。这让你可以给用户一个关于当前进度的文本描述。

这个 `Update()` 方法通常返回 `True` 。但是，如果用户通过 `Cancel` 按钮已经取消了该对话框，那么下次的 `Update()` 将返回 `False` 。这是你响应用户的取消请求的机会。要检测用户的取消请求，我们建议你尽可能频繁地 `Update()` 。

使用标准对话框

大多数操作系统都为像文件选择、字体选择和颜色选择这些任务提供了标准对话框。这为平台提供了一致感观。你也可以使用来自于 `wxPython` 的这些对话框，它们也为你的应用程序提供了一致的感观。如果你使用 `wxPython` ，那么它为你提供了类似的对话框，即使所在的平台没有提供系统对话框。

如何使用文件选择对话框？

在 `wxPython` 中， `wx.FileDialog` 为主流的平台使用本地操作系统对话框，对其它操作系统使用非本地相似的外观。微软 `Windows` 的版本如图9.6所示。

图**9.6**

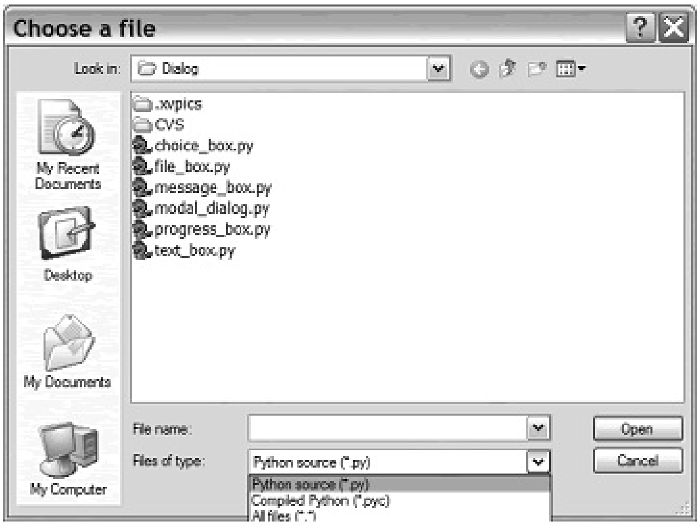


Figure 9.6
The standard Windows file chooser

你可以设置文件对话框开始在任一目录，你也可以使用通配符过滤来限制去显示某种文件类型。例9.6显示了一个基本的例子。

例**9.6** 使用 `wx.FileDialog`

```
import wx
import os

if __name__ == "__main__":
    app = wx.PySimpleApp()
    wildcard = "Python source (*.py)|*.py|" \
            "Compiled Python (*.pyc)|*.pyc|" \
            "All files (*.*)|*.*"
dialog = wx.FileDialog(None, "Choose a file", os.getcwd(),
            "", wildcard, wx.OPEN)
    if dialog.ShowModal() == wx.ID_OK:
print dialog.GetPath()

    dialog.Destroy()
```

文件对话框是我们这章已见过的最复杂的对话框，它有几个属性可以通过编程的方式读写。它的构造函数使得你能够设置它的一些属性：

```
wx.FileDialog(parent, message="Choose a file", defaultDir="",
defaultFile="", wildcard="*.*", style=0,
        pos=wx.DefaultPosition)
```

`message` 参数出现在窗口的标题栏中。 `defaultDir` 参数告诉对话框初始的时候显示哪个目录。如果这个参数为空或表示的目录不存在，那么对话框开始在当前目录。 `defaultFile` 是默认保存为的文件。 `wildcard` 参数使你可以基于给定的模式来过滤列表，使用通常的和?作为通配符。通配符可以是单个模式，如. `py` 或格式如 描述 | 模式 | 描述 | 模式 的一系列模式——类似于例9.6中所用。

" `Python source` (. *py)* |. `py` | `Compiled Python` (. *pyc)* |. `pyc` |

- `All files` (.)|."

如果有一个多个项目的模式，那么它们显示在图9.6所示的下拉菜单中。 `pos` 参数不保证被基本的系统所支持。

选择一个文件

`wx.FileDialog` 的两个最重要的样式标记是 `wx.OPEN` 和 `wx.SAVE` ，它们表明对话框的类型并影响对话框的行为。

用于打开文件的对话框有两个标记，它们进一步影响对话框的行为。 `wx.HIDE_READONLY` 标记灰化复选框，使用户以只读模式打开文件。 `wx.MULTIPLE` 标记使用户可以在一个目录中选择打开多个文件。

保存文件对话框有一个有用的标记 `wx.OVERWRITE_PROMPT` ，它使得保存文件时，如果有相同的文件存在，则提示用户是否覆盖。

两种文件对话框都可以使用 `wx.CHANGE_DIR` 标记。当使用这个标记时，文件的选择也可改变应用程序的工作目录为所选文件所在的目录。这使得下次文件对话框打开在相同的目录，而不需要应用程序再在别处存储该值。

和本章迄今为止我人们所见过的其它对话框不一样，文件对话框的属性 `directory` ， `filename` , `style` , `message` , 和 `wildcard` 是可以通过方法来得到和设置的。这些方法使用 `Get` / `Set` 命名习惯。

在用户退出对话框后，如果返回值是 `wx.OK` ，那么你可以使用方法 `GetPath()` 来得到用户的选择，该函数的返回值是字符串形式的文件全路径名。如果对话框是一个使用了 `wx.MULTIPLE` 标记的打开对话框，则用 `GetPaths()` 代替 `GetPath()` 。该方法返回路径字符串的一个 `Python` 列表。如果你需要知道在用户选择时使用了下拉菜单中的哪个项，你可以使用 `GetFilterIndex()` ，它返回项目的索引。要通过编程改变索引，使用方法 `SetFilterIndex()` 。

这后面的是一个使用文件对话框的便利函数：

```
wx.FileSelector(message, default_path="", default_filename="",
    default_extension="", wildcard="*.*'', flags=0, parent=None,
    x=-1, y=-1)
```

`message` , `default_path` , `default_filename` , 和 `wildcard` 参数意义与构造函数的基本相同，尽管参数的名字不同。 `flags` 参数通常被称作 `style` ， `default_extension` 参数是保存为文件时默认的后缀（如果用户没有指定后缀的情况下）。如果用户按下 `OK` ，返回值是字符串形式的路径名，如果用户按下 `Cancel` 则返回一个空字符串。

选择一个目录

如果用户想去选择一个目录而非一个文件，使用 `wx.DirDialog` ，它呈现一个目录树的视图，如图9.7所示。

这个目录选择器比文件对话框简单些。例9.7显示了相关的代码。

例**9.7** 显示一个目录选择对话框

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.DirDialog(None, "Choose a directory:",
          style=wx.DD_DEFAULT_STYLE | wx.DD_NEW_DIR_BUTTON)
    if dialog.ShowModal() == wx.ID_OK:
        print dialog.GetPath()
    dialog.Destroy()
```

图**9.7**

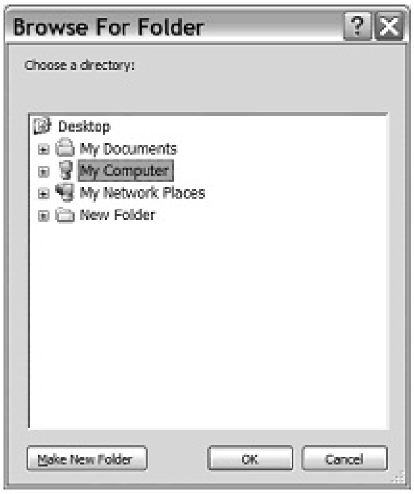


Figure 9.7
A directory selection dialog

这个对话框的所有的功能几乎都在构造函数中：

```
wx.DirDialog(parent, message="Choose a directory", defaultPath="
",
style=0, pos = wx.DefaultPosition, size = wx.DefaultSize,
name="wxDirCtrl")
```

由于 `message` 参数显示在对话框中，所以你不需要一个钩子去改变标题栏。 `defaultPath` 告诉对话框选择的默认路径，如果它为空，那么对话框显示文件系统的根目录。 `pos` 和 `size` 参数在微软 `Windows` 下被忽略， `name` 参数在所有的操作系统下都被忽略。该对话框的样式标记 `wx.DD_NEW_DIR_BUTTON` 给对话框一个用于创建目录的一个按钮。这个标记在老版的微软 `Windows` 中不工作。

`wx.DirDialog` 类的 `path` , `message` ,和 `style` 属性都有相应的 `get` 和 *`set`* 方法。你可以使用 `GetPath()` 方法来在对话框被调用后获取用户的选择。这个对话框也有一个便利的函数：

```
wx.DirSelector(message=wx.DirSelectorPromptStr, default_path="",
    style=0, pos=wxDefaultPosition, parent=None)
```

所有的参数和前面的构造函数相同。如果 `OK` 被按下，则该函数返回所选择的字符串形式的目录名，如果按下 `Cancel` ，则返回空字符串。

如何使用字体选择对话框？

在 `wxPython` 中，字体选择对话框与文件对话框是不同的，因为它使用了一个单独的帮助类来管理它所呈现的信息。图9.8显示了微软 `Windows` 版的字体对话框。

例9.8 显示了产生图9.8的代码，并且与前面的对话框例子看起来也有些不同。

例**9.8** 字体对话框

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.FontDialog(None, wx.FontData())
    if dialog.ShowModal() == wx.ID_OK:
        data = dialog.GetFontData()
        font = data.GetChosenFont()
        colour = data.GetColour()
        print 'You selected: "%s", %d points\n' % (
                font.GetFaceName(), font.GetPointSize())
    dialog.Destroy()
```

图**9.8**

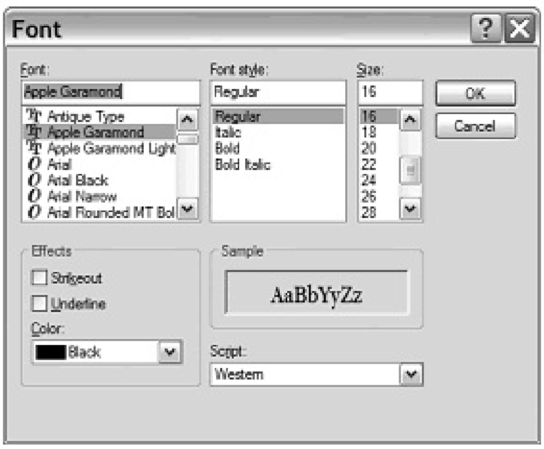


Figure 9.8
A sample font picker dialog

`wx.FontDialog` 的构造函数比前面的那些简单的多：

`wx.FontDialog(parent` , `data)`

你不能为该对话框设置一个消息或标题，并且被通常作为样式标记传递的信息被包含在`data`参数中，该参数是类`wx.FontData`。`wx.FontData`类自己只有一个有用的方法：`GetFontData()`，该方法返回字体数据的实例。

`wx.FontData`的实例使你能够设置管理字体对话框显示的值，并且也能够容纳用户输入的信息。例如，在例9.8中的代码调用了`wx.FontData`实例的两个`get`*方法来确定所选字体的细节。`wx.FontData`的构造函数没有参数——所有的属性必须通过使用表9.4中的方法来设置。

表**9.4** **`wx.FontData`**的方法

`GetAllowSymbols()`，`SetAllowSymbols(allowSymbols)`	决定是否在对话示符号字体（如 `dingbats` 数是布尔值。只在 `Windows` 中有该属性的初始值是 `True` 。
`GetChosenFont()` ，`SetChosenFont(font)`	以 `wx.Font` 对象返回用户所选的果用户选择了取返回 `None` 。 `wx.` 将在第12章作更讨论。
`GetColour()`，`SetColour(colour)`	返回在对话框的部分所选的颜色。 `set` 方法使预先设定默认值。 *`get`* 方法返个 `wx.Colour` 实例。 `set` *方法的 `colour` 只能个 `wx.Colour` 或色的字符串名。初始值是 `black`
`GetEnableEffects()` ，`EnableEffects(enable)`	在该对话框的 `Windows` 版本属性控制是否显所选颜色、中间线通过、是否带特性。
`GetInitialFont()`，`SetInitialFont(font)`	返回对话框初值（即当前所用的这个属性可以在示之前通过应用的来设置。它的是 `None` 。
`SetRange(min ， max)`	设置字体尺寸（效范围。仅用于的 `Windows` 系统值是0~0，意味没的限制。

|| `GetShowHelp()` , `SetShowHelp()` ||如果为 `True` ，那么该对话框的微软 `Windows` 版本将显示一个帮助按钮。初始值为 `False` ||。

有一个使用字体对话框的便利的函数，它回避了 `wx.FontData` 类：

```
wx.GetFontFromUser(parent, fontInit)
```

`fontInit` 参数是 `wx.Font` 的一个实例，它用作对话框的初始值。该函数的返回值是一个 `wx.Font` 实例。如用户通过 `OK` 关闭了对话框，则方法 `wx.Font.Ok()` 返回 `True` ，否则返回 `False` 。

如何使用颜色对话框？

颜色对话框类似于字体对话框，因为它使用了一个外部的数据类来管理它的信息。图9.9显示了这类对话框的微软版本。

例9.9显示了生成该对话框的代码，它几乎与前面的字体对话框相同。

例**9.9**

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.ColourDialog(None)
    dialog.GetColourData().SetChooseFull(True)
    if dialog.ShowModal() == wx.ID_OK:
        data = dialog.GetColourData()
        print 'You selected: %s\n' % str(data.GetColour().Get())
    dialog.Destroy()
```

图**9.9**

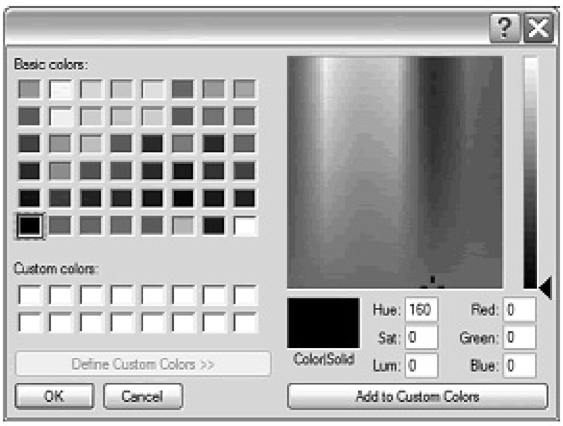


Figure 9.9
A standard wxPython color picker

用于颜色选择器的 `wxPython` 的类是 `wx.ColourDialog` 。它的构造函数很简单，没有太多的参数：

`wx.ColourDialog(parent` , `data` = `None)`

`data` 参数是类 `wx.ColourData` 的实例，它比相应字体的更简单。它只包含默认的没有参数的构造函数和后面的三个属性：

1、`GetChooseFull` / `SetChooseFull(flag)` ：仅在微软 `Windows` 下工作。当设置后，将显示完整的对话框，包括自定义颜色选择器。如果不设置，则自定义颜色选择器不被显示。

2、`GetColour` / `SetColour(colour)` ：当图表被关闭后，调用 `get` *来看用户的选择。最初它被设置为 `black` 。如果在对话框显示之前设置了它，那么对话框最初显示为该颜色。

3、`GetCustomColour(i)` / `SetCustomColour(i` , `colour)` ：根据自定义的颜色数组中的索引i来返回或设置元素。i位于[0,15]之间。初始时，所有的自定义颜色都是白色。

一个回避了 `wx.ColorData` 的使用颜色对话框的便利函数是：

`wx.GetColourFromUser(parent` , `colInit)`

`colInit` 是 `wx.Colour` 的一个实例，并且当对话框显示时它是对话框的初始的值。函数的返回值也是一个 `wx.Colour` 的实例。如果用户通过 `OK` 关闭了对话框，那么方法 `wx.Colour.OK()` 返回 `True` 。如果用户通过 `Cancel` 关闭了对话框，那么方法 `wx.Colour.OK()` 返回 `False` 。

如何使用户能够浏览图像？

如果你在你的程序中做图形处理，那么在他们浏览文件树时使用缩略图是有帮助的。用于该目的的 `wxPython` 对话框被称为 `wx.lib.imagebrowser.ImageDialog` 。图9.10显示了一个例子。

例9.10显示了用于该图像浏览对话框的简单的代码。

例**9.10** 创建一个图像浏览对话框

```
import wx
import wx.lib.imagebrowser as imagebrowser

if __name__ == "__main__":
    app = wx.PySimpleApp()
dialog = imagebrowser.ImageDialog(None)
    if dialog.ShowModal() == wx.ID_OK:
print "You Selected File: " + dialog.GetFile()
    dialog.Destroy()
```

图**9.10**

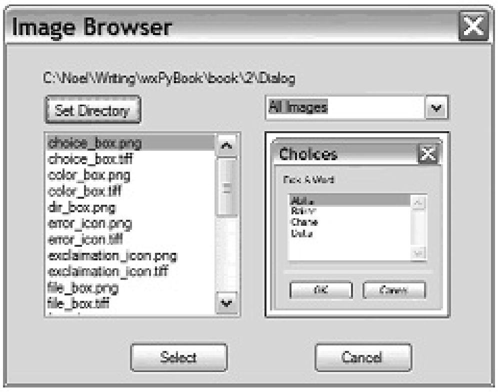


Figure 9.10 A typical image dialog browser

`wx.lib.imagebrowser.ImageDialog` 类是十分简单的，并有相对较少的选项供程序员去设置。要改变该对话框的行为的话，请查阅改变显示的文件类型的 `Python` 源码。类的构造函数要求两个参数。

`ImageDialog(parent` , `set_dir` = `None)`

`set_dir` 参数是对话框显示时所在的目录。如果不设置，那么使用应用程序当前的工作目录。在对话框被关闭后， `GetFile()` 返回所选文件的完整路径字符串， `GetDirectory()` 只返回目录部分。

创建向导

向导是一系列被链接在一起的简单对话框，它使得用户一步步地跟随它们。通常它们被用于指导用户的安装或一个复杂的配置过程。图9.11显示了一个向导示例。

图**9.11**



Figure 9.11 A simple wizard sample

在 `wxPython` 中，一个向导是一系列的页面，它由类 `wx.wizard.Wizard` 的一个实例控制。向导实例管理用户的页面切换事件。这些页面自身也是类 `wx.wizard.WizardPageSimple` 或 `wx.wizard.WizardPage` 的实例。这两种

类的实例，它们只不过是附加了必要的管理页面链接逻辑的 `wx.Panel` 的实例。已证明这两个实例之间的区别仅当用户按下 `Next` 按钮
时。 `wx.wizard.WizardPage` 的实例使你能够动态地决定浏览哪页，
而 `wx.wizard.WizardPageSimple` 的实例要求向导被显示前，顺序被预先设置。例9.11显示了产生图9.11的代码。

例**9.11** 创建一个简单的静态向导

```
import wx
import wx.wizard

class TitledPage(wx.wizard.WizardPageSimple):#1 创建页面样板
    def __init__(self, parent, title):
        wx.wizard.WizardPageSimple.__init__(self, parent)
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.SetSizer(self.sizer)
        titleText = wx.StaticText(self, -1, title)
        titleText.SetFont(
                wx.Font(18, wx.SWISS, wx.NORMAL, wx.BOLD))
        self.sizer.Add(titleText, 0,
                wx.ALIGN_CENTRE | wx.ALL, 5)
        self.sizer.Add(wx.StaticLine(self, -1), 0,
                wx.EXPAND | wx.ALL, 5)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    wizard = wx.wizard.Wizard(None, -1, "Simple Wizard")# 创建向
导实例

    # 创建向导页面
    page1 = TitledPage(wizard, "Page 1")
    page2 = TitledPage(wizard, "Page 2")
    page3 = TitledPage(wizard, "Page 3")
    page4 = TitledPage(wizard, "Page 4")

    page1.sizer.Add(wx.StaticText(page1, -1,
            "Testing the wizard"))
    page4.sizer.Add(wx.StaticText(page4, -1,
            "This is the last page."))

    #2 创建页面链接
    wx.wizard.WizardPageSimple_Chain(page1, page2)
    wx.wizard.WizardPageSimple_Chain(page2, page3)
    wx.wizard.WizardPageSimple_Chain(page3, page4)

    wizard.FitToPage(page1)#3 调整向导的尺寸

    if wizard.RunWizard(page1):#4 运行向导
        print "Success"

    wizard.Destroy()
```

#1 为了便于移植的目的，我们创建了一个简单的小的页面，它包含了一个静态文本标题。通常情况下，这儿还包含一些表单元素，可能还有一些要用户输入的数据。

#2 `wx.wizard.WizardPageSimple_Chain()` 函数是一个便利的方法，它以两个页面为参数相互地调用它们的 `SetNext()` 和 `SetPrev()` 方法。

#3 `FitToSize()` 根据页面参数及该页链条上的所有页面调整向导的大小。该方法只能在页面链接被创建后调用。

#4 该方法的参数是向导开始时的页面。向导在它到达一个没有下一页的页面时知道去关闭。如果用户浏览了整个向导并通过按下 `Finish` 按钮退出的话， `RunWizard()` 方法返回 `True` 。

创建 `wx.wizard.Wizard` 实例是使用向导的第一步。其构造函数如下：

```
wx.wizard.Wizard(parent, id=-1, title=wx.EmptyString,
bitmap=wx.NullBitmap, pos=wx.DefaultPosition)
```

在这里的 `parent` , `id` , `title` , `pos` 和 `wx.Panel` 的意义相同。如果设置了 `bitmap` 参数，那么该参数将显示在每一页上。这儿只有一个样式标记： `wx.wizard.WIZARD_EX_HELPBUTTON` ，它显示一个帮助按钮。这是一个扩展的标记，需要使用第8章所说的两步创建过程。

通常，你将调用例9.11的#3中所示的 `FitToSize()` 来管理窗口的尺寸，但是你也可以通过调用带有一个元组或 `wx.Size` 实例的 `SetPageSize()` 来设置一个最小的尺寸。 `GetPageSize()` 方法返回当前的尺寸，在这两种情况下，该尺寸仅用于对话框中的单个的页面部分，而作为一个整体的对话框将更大一些。

你可以管理该类中的页面。方法 `GetCurrentPage()` 返回当前被显示的页面，如果该向导当前没有被显示，该方法返回 `None` 。你可以通过调用 `HasNextPage()` 和 `HasPrevPage()` 来确定当前页是否有下一页或上一页。 使用 `RunWizard()` 来运行该向导，如例9.11中#4的说明。

向导产生的命令事件如表9.5所示，你可以捕获这些事件，以便作专门的处理。这些事件对象属于类 `wx.wizard.WizardEvent` ，它们提供了两个可用的方法。 `GetPage()` 返回 `wx.WizardPage` 的实例，该实例在向导事件产生时是有效，而非作为事件结果被显示的实例。如果事件是前进一页，那么 `GetDirection()` 返回 `True` ，如果事件是后退一页，那么 `GetDirection()` 返回 `False` 。

表9.5 `wx.wizard.WizardDialog` 的事件

EVT_WIZARD_CANCEL	当用户按下 Cancel 按钮时产生。该事件可以使用 Veto() 来否决，这种情况下，对话框将不会消失。
EVT_WIZARD_FINISHED	当用户按下 Finish 按钮时产生。
EVT_WIZARD_HELP	当用户按下 Help 按钮时产生。
EVT_WIZARD_PAGE_CHANGED	在页面被切换后产生。
EVT_WIZARD_PAGE_CHANGING	当用户已请求了一个页面切换时产生，这时页面还没有发生切换。这个事件可以被否决（例如，如果页面上有一个必须被填写的字段）。

wx.wizard.WizardPageSimple 类被当作一个面板一样。它的构造函数使你可以设置上一页和下一页，如下所示：

wx.wizard.WizardPageSimple(parent = None , prev = None , next = None)

如果你想在构造器中设置它们，你可以使用 SetPrev() 和 SetNext() 方法。如果那样太麻烦，你可以使用 wx.wizard.WizardPageSimple_Chain() ，它设置两页间的链接关系。

向导页的复杂版： wx.wizard.WizardPage ，稍微不同。它没有显式地设置前一页和下一页，而是使你能够使用更复杂的逻辑去定义下一步到哪儿。它的构造函数如下：

```
wx.WizardPage(parent, bitmap=wx.NullBitmap, resource=None)
```

如果 bitmap 参数被设置了，那么该参数覆盖父向导中所设置的位图。 resource 参数从一个 wxPython 资源装载页面。要处理页面逻辑，就要覆盖 GetPrev() 和 GetNext() 方法来返回你想要向导下一步的位置。该类的一个典型的用法是根据用户对当前页的响应动态地决定接下来的页面。

显示启动提示

许多应用程序都使用启动提示来作为一种向用户介绍该程序的特性等信息的方法。在 wxPython 中有一个非常简单的机制用来显示启动提示。图9.12显示了一个提示窗口的示例。

图**9.12**



Figure 9.12 A sample tip window with a helpful message.

例9.12显示了相关代码

例**9.12**

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    provider = wx.CreateFileTipProvider("tips.txt", 0)
    wx.ShowTip(None, provider, True)
```

有两个便利的函数用来管理启动提示。第一个如下创建一个 `wx.TipProvider` ：

```
wx.CreateFileTipProvider(filename, currentTip)
```

`filename` 是包含提示字符串的文件的名字。 `currentTip` 是该文件中用于一开始显示的提示字符的索引，该文件中的第一个提示字符串的索引是0。

提示文件是一个简单的文本文件，其中的每一行是一个不同的提示。空白行被忽略，以#开始的行被当作注释并也被忽略。下面是上例所使用的提示文件中的内容：

`You` `can` `do` `startup` `tips` `very` `easily.` `Feel` `the` `force` , `Luke.`

提示的提供者（ `provider` ）是类 `wx.PyTipProvider` 的一个实例。如果你需要更细化的功能，你可以创建你自己的 `wx.TipProvider` 的子类并覆盖 `GetTip()` 函数。

显示提示的函数是 `wx.ShowTip():`

```
wx.ShowTip(parent, tipProvider, showAtStartup)
```

`parent` 是父窗口，`tipProvider` 通常创建自 `wx.CreateFileTipProvider` 。`showAtStartup` 控制启动提示显示时，复选框是否被选择。该函数的返回值是复选框的状态值，以便你使用该值来决定你的应用程序下次启动时是否显示启动提示。

使用验证器（validator）来管理对话框中的数据

验证器是一个特殊的 `wxPython` 对象，它简化了对话框中的数据管理。当我们在第三章中讨论事件时，我们简要的提及到如果一个窗口部件有一个验证器，那么该验证器能够被事件系统自动调用。我们已经见过了几个 `wxPython` 窗口部件类的构造函数中将验证器作为参数，但是我们还没有讨论它们。

验证器有三个不相关的功能：

1、在对话框关闭前验证控件中的数据 2、自动与对话框传递数据 3、验证用户键入的数据

=== 如何使用验证器来确保正确的?譁苠?===

验证器对象是 `wx.Validator` 的子类。父类是抽象的，不能直接使用。尽管在C++ `wxWidget` 集中有一对预定义的验证器类，但是在 `wxPython` 中，你需要定义你自己的验证器类。正如我们在别处所见的，你的 `Python` 类需要继承自 `Python` 特定的子类：`wx.PyValidator` ，并且能够覆盖该父类的所有方法。一个自定义的验证器子类必须覆盖方法 `Clone()` ，该方法应该返回验证器的相同的副本。

一个验证器被关联到你的系统中的一个特定的窗口部件。这可以用两种方法之一来实现。第一种方法，如果窗口部件许可的话，验证器可以被作为一个参数传递给该窗口部件的构造函数。如果该窗口部件的构造函数没有一个验证器参数，你仍然可以通过创建一个验证器实例并调用该窗口部件的 `SetValidator(validator)` 方法来关联一个验证器。

要验证控件中的数据，你可以先在你的验证器子类中覆盖 `Validate(parent)` 方法。`parent` 参数是验证器的窗口部件的父窗口（对话框或面板）。如果必要，可以使用这个来从对话框中其它窗口部件得到数据，或者你可以完全忽略该参数。你可以使用 `self.GetWindow()` 来得到正在被验证的窗口部件的一个引用。你的 `Validate(parent)` 方法的返回值是一个布尔值。`True` 值表示验证器的窗口部件中的数据已验证了。`False` 表示有问题。你可以根据 `Validate()` 方法来使用 `x.MessageBox()` 去显示一个警告，但是你不应该做其它的任何可以在 `wxPython` 应用程序中引发事件的事情。

`Validate()` 的返回值是很重要的。它在你使用 `OK` 按钮（该按钮使用 `wx.ID_OK` `ID` ）企图关闭一个对话框时发挥作用。作为对 `OK` 按钮敲击处理的一部分，`wxPython` 调用对话框中有验证器的窗口部件的 `Validate()` 函数。如果任一验证器返回 `False` ，那么对话框不将关闭。例9.13显示了一个带有验证器的示例对话框，它检查所有文本控件中有无数据。

例**9.13** 检查所有文本控件有无数据的验证器

```
import wx

about_txt = """\
The validator used in this example will ensure that the text
controls are not empty when you press the Ok button, and
will not let you leave if any of the Validations fail."""

class NotEmptyValidator(wx.PyValidator):# 创建验证器子类
     def __init__(self):
         wx.PyValidator.__init__(self)

     def Clone(self):
         """
 Note that every validator must implement the Clone() method.
 """
         return NotEmptyValidator()

     def Validate(self, win):#1 使用验证器方法
         textCtrl = self.GetWindow()
         text = textCtrl.GetValue()

         if len(text) == 0:
             wx.MessageBox("This field must contain some text!",
 "Error")
             textCtrl.SetBackgroundColour("pink")
             textCtrl.SetFocus()
             textCtrl.Refresh()
             return False
         else:
             textCtrl.SetBackgroundColour(
                 wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDO
W))
             textCtrl.Refresh()
             return True

     def TransferToWindow(self):
return True

     def TransferFromWindow(self):
         return True

class MyDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, "Validators: validati
ng")

        # Create the text controls
        about   = wx.StaticText(self, -1, about_txt)
        name_l  = wx.StaticText(self, -1, "Name:")
        email_l = wx.StaticText(self, -1, "Email:")
        phone_l = wx.StaticText(self, -1, "Phone:")
```

```
        #2 使用验证器
        name_t  = wx.TextCtrl(self, validator=NotEmptyValidator(
))
        email_t = wx.TextCtrl(self, validator=NotEmptyValidator(
))
        phone_t = wx.TextCtrl(self, validator=NotEmptyValidator(
))

        # Use standard button IDs
        okay   = wx.Button(self, wx.ID_OK)
        okay.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        # Layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(about, 0, wx.ALL, 5)
        sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

        fgs = wx.FlexGridSizer(3, 2, 5, 5)
        fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(name_t, 0, wx.EXPAND)
        fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(email_t, 0, wx.EXPAND)
        fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(phone_t, 0, wx.EXPAND)
        fgs.AddGrowableCol(1)
        sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

        btns = wx.StdDialogButtonSizer()
        btns.AddButton(okay)
        btns.AddButton(cancel)
        btns.Realize()
        sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

        self.SetSizer(sizer)
        sizer.Fit(self)

app = wx.PySimpleApp()

dlg = MyDialog()
dlg.ShowModal()
dlg.Destroy()

app.MainLoop()
```

#1 该方法测试基本的控件有无数据。如果没有，相应的控件的背景色变为粉红色。

#2 这几行，对话框中的每个文本控件都关联一个验证器。

图9.13显示了一个文本域为空就企图关闭的对话框。

图**9.13**

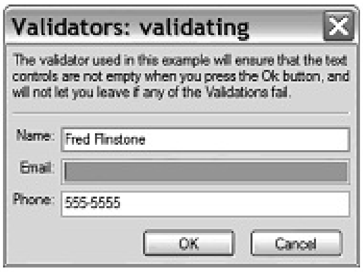


Figure 9.13 Attempting to close an invalid validator

明确地告诉对话框去核对验证器的代码没有出现在示例中，因为它是 `wxPython` 事件系统的一部分。对话框与框架之间的另一区别是对话框有内建的验证器行为，而框架没有。如果你喜欢将验证器用于不在对话框内的控件，那么调用父窗口的 `Validate()` 方法。如果父窗口已设置了 `wx.WS_EX_VALIDATE_RECURSIVELY` 额外样式，那么所有的子窗口的 `Validate()` 方法也被调用。如果任一验证失败，那么 `Validate` 返回 `False` 。接下来，我们将讨论如何将验证器用于数据传输。

如何使用验证器传递数据？

验证器的第二个重要的功能是，当对话框打开时，它自动将数据传送给对话框显示，当该对话框关闭时，自动从对话框把数据传输到一个外部资源。图9.14显示了一个示例对话框。

图**9.14**

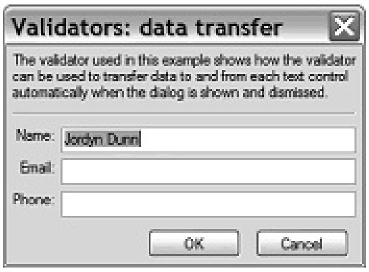


Figure 9.14 The transferring validator—this dialog will automatically display entered values when closed

要实现这个，你必须在你的验证器子类中覆盖两个方法。方法 `TransferToWindow()` 在对话框打开时自动被调用。你必须使用这个方法把数据放入有验证器的窗口部件。 `TransferFromWindow()` 方法在使用 `OK` 按钮关闭对话框窗口时且数据已被验证后被自动调用。你必须使用这个方法来将数据从窗口部件移动给其它的资源。

数据传输必须发生的事实意味着验证器必须对一个外部的数据对象有一些了解，如例9.14所示。在这个例子中，每个验证器都使用一个全局数据字典的引用和一个字典内的对于相关控件重要的关键字来被初始化。

当对话框打开时， `TransferToWindow()` 方法从字典中根据关键字读取数据并把数据放入文本域。当对话框关闭时， `TransferFromWindow()` 方法反向处理并把数据写入字典。这个例子的对话框显示你传输的数据。

例**9.14** 一个数据传输验证器

```
import wx
import pprint

about_txt = """\
The validator used in this example shows how the validator
can be used to transfer data to and from each text control
automatically when the dialog is shown and dismissed."""

class DataXferValidator(wx.PyValidator):# 声明验证器
     def __init__(self, data, key):
         wx.PyValidator.__init__(self)
         self.data = data
         self.key = key

     def Clone(self):
         """
 Note that every validator must implement the Clone() method.
 """
         return DataXferValidator(self.data, self.key)

     def Validate(self, win):# 没有验证数据
         return True

     def TransferToWindow(self):# 对话框打开时被调用
         textCtrl = self.GetWindow()
         textCtrl.SetValue(self.data.get(self.key, ""))
return True

     def TransferFromWindow(self):# 对话框关闭时被调用
         textCtrl = self.GetWindow()
         self.data[self.key] = textCtrl.GetValue()
         return True

class MyDialog(wx.Dialog):
    def __init__(self, data):
        wx.Dialog.__init__(self, None, -1, "Validators: data tra
nsfer")

        # Create the text controls
        about   = wx.StaticText(self, -1, about_txt)
        name_l  = wx.StaticText(self, -1, "Name:")
        email_l = wx.StaticText(self, -1, "Email:")
        phone_l = wx.StaticText(self, -1, "Phone:")

        # 将验证器与窗口部件相关联
        name_t  = wx.TextCtrl(self, validator=DataXferValidator(
```

```
data, "name"))
        email_t = wx.TextCtrl(self, validator=DataXferValidator(
data, "email"))
        phone_t = wx.TextCtrl(self, validator=DataXferValidator(
data, "phone"))

        # Use standard button IDs
        okay   = wx.Button(self, wx.ID_OK)
        okay.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        # Layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(about, 0, wx.ALL, 5)
        sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

        fgs = wx.FlexGridSizer(3, 2, 5, 5)
        fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(name_t, 0, wx.EXPAND)
        fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(email_t, 0, wx.EXPAND)
        fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(phone_t, 0, wx.EXPAND)
        fgs.AddGrowableCol(1)
        sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

        btns = wx.StdDialogButtonSizer()
        btns.AddButton(okay)
        btns.AddButton(cancel)
        btns.Realize()
        sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

        self.SetSizer(sizer)
        sizer.Fit(self)

app = wx.PySimpleApp()

data = { "name" : "Jordyn Dunn" }
dlg = MyDialog(data)
dlg.ShowModal()
dlg.Destroy()

wx.MessageBox("You entered these values:\n\n" +
              pprint.pformat(data))

app.MainLoop()
```

对话框中验证器的传输数据方法的调用自动发生。要在非对话框窗口中使用验证器来传输数据，必须调用父窗口部件的 `TransDataFromWindow()` 和 `TransferDataToWindow()` 方法。如果该窗口设

置了 `wx.WS_EX_VALIDATE_RECURSIVELY` 额外样式，那么在所有的子窗口部件上也将调用该传输函数。

如何在数据被键入时验证数据？

在数据被传给窗口部件之前，你也可使用验证器来在用户输入数据时验证所输入的数据。这是非常有用的，因为它可以防止将得到的坏的数据传入你的应用程序。图9.12显示了一个例子，其中对话框的文本阐明了该思想。

图**9.15**



Figure 9.15 A validator verifying data on the fly

验证数据的方法的自动化成份少于其它的机制。你必须显式绑定验证器的窗口部件的字符事件给一个函数，如下所示：

`self.Bind(wx.EVT_CHAR` , `self.OnChar)`

该窗口部件假设事件源属于验证器。例9.15显示了这个绑定。

例**9.15** 实时验证

```
import wx
import string

about_txt = """\
The validator used in this example will validate the input on th
e fly
instead of waiting until the okay button is pressed.  The first
field
will not allow digits to be typed, the second will allow anythin
g
and the third will not allow alphabetic characters to be entered
.
"""

class CharValidator(wx.PyValidator):
    def __init__(self, flag):
         wx.PyValidator.__init__(self)
         self.flag = flag
         self.Bind(wx.EVT_CHAR, self.OnChar)# 绑定字符事件
```

```
    def Clone(self):
         """
 Note that every validator must implement the Clone() method.
 """
         return CharValidator(self.flag)

    def Validate(self, win):
         return True

    def TransferToWindow(self):
return True

    def TransferFromWindow(self):
         return True

    def OnChar(self, evt):# 数据处理
         key = chr(evt.GetKeyCode())
         if self.flag == "no-alpha" and key in string.letters:
              return
         if self.flag == "no-digit" and key in string.digits:
              return
         evt.Skip()

class MyDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, "Validators: behavior
 modification")

        # Create the text controls
        about   = wx.StaticText(self, -1, about_txt)
        name_l  = wx.StaticText(self, -1, "Name:")
        email_l = wx.StaticText(self, -1, "Email:")
        phone_l = wx.StaticText(self, -1, "Phone:")

        # 绑定验证器
        name_t  = wx.TextCtrl(self, validator=CharValidator("no-
digit"))
        email_t = wx.TextCtrl(self, validator=CharValidator("any
"))
        phone_t = wx.TextCtrl(self, validator=CharValidator("no-
alpha"))

        # Use standard button IDs
        okay   = wx.Button(self, wx.ID_OK)
        okay.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        # Layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(about, 0, wx.ALL, 5)
        sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

        fgs = wx.FlexGridSizer(3, 2, 5, 5)
```

```
        fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(name_t, 0, wx.EXPAND)
        fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(email_t, 0, wx.EXPAND)
        fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(phone_t, 0, wx.EXPAND)
        fgs.AddGrowableCol(1)
        sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

        btns = wx.StdDialogButtonSizer()
        btns.AddButton(okay)
        btns.AddButton(cancel)
        btns.Realize()
        sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

        self.SetSizer(sizer)
        sizer.Fit(self)

app = wx.PySimpleApp()

dlg = MyDialog()
dlg.ShowModal()
dlg.Destroy()

app.MainLoop()
```

由于 `OnChar()` 方法是在一个验证器中，所以它在窗口部件响应字符事件之间被调用。该方法让你可以通过使用 `Skip()` 来将事件传送给窗口部件。你必须调用 `Skip()` ，否则验证器将妨碍正常的事件处理。验证器执行一个测试来查看用于该控件的字符是否有效。如果该字符无效，那么 `Skip()` 不被调用，并且事件处理停止。如果有必须的话，除了 `wx.EVT_CHAR` 之外的其它事件也可以被绑定，并在窗口部件响应之前验证器处理那些事件。

对于处理你 `wxPython` 应用程序中的数据，验证器是一个强大且灵活的机制。适当地使用它们，可以让你的应用程序的开发和维护更加的顺畅。

本章小结

1、对话框被用于在有一套特殊的信息需要被获取的情况下，处理与用户的交互，这种交互通常很快被完成。在 `wxPython` 中，你可以使用通用的 `wx.Dialog` 类来创建你自己的对话框，或者你也可以使用预定义的对话框。大多数情况下，通常被使用的对话框都有相应的便利函数，使得这种对话框的使用更容易。

2、对话框可以显示为模式对话框，这意味在对话框可见时，该应用程序中的用户所有的其它输入将被阻塞。模式对话框通过使用 `ShowModal()` 方法来被调用，它的返回值依据用户所按的按钮而定（ `OK` 或 `Cancel` ）。关闭模式对话框并不会销毁它，该对话框的实例可以被再用。

3、在 `wxPython` 中有三个通用的简单对话框。 `wx.MessageDialog` 显示一个消息对话框。 `wx.TextEntryDialog` 使用户能够键入文本， `wx.SingleChoiceDialog` 给用户一个基于列表项的选择。

4、当正在执行一个长时间的后台的任务时，你可以使用 `wx.ProgressDialog` 来给用户显示进度信息。用户可以通过 `wx.FileDialog` 使用标准文件对话框来选择一个文件。可以使用 `wx.DirDialog` 来创建一个标准目录树，它使得用户可以选择一个目录。

5、你可以使用 `wx.FontDialog` 和 `wx.ColorDialog` 来访问标准的字体选择器和颜色选择器。在这两种情况中，对话框的行为和用户的响应是由一个单独的数据类来控制的。

6、要浏览缩略图，可以使用 `wxPython` 特定的类 `wx.lib.imagebrowser.ImageDialog` 。这个类使用户能够通过文件系统并选择一个图像。

7、你可以通过使用 `wx.wizard.Wizard` 创建一个向导来将一组相关的对话框表单链接起来。对话框表单是 `wx.wizard.WizardSimplePage` 或 `wx.wizard.WizardPage` 的实例。两者的区别是， `wx.wizard.WizardSimplePage` 的页到页的路线需要在向导被显示之前就安排好，而 `wx.wizard.WizardPage` 使你能够在运行时管理页到页的路线的逻辑。

8、使用 `wx.CreateFileTipProvider` 和 `wx.ShowTip` 函数可以很容易地显示启动提示。

9、验证器是很有用的对象，如果输入的数据不正确的话，它可以自动阻止对话框的关闭。他们也可以在一个显示的对话框和一个外部的对象之间传输数据，并且能够实时地验证数据的输入。

第十章 创建和使用wxPython菜单

本章内容包括：

创建菜单 使用菜单项工作 添加子菜单、弹出菜单和自定义菜单 菜单的设计准则

难以想象一个应用程序的顶部没有我们常见的以 `File` 和 `Edit` 开头，以 `Help` 结尾的栏目。这太糟糕了。菜单是那些隐藏在背后并不太重视绘制的标准界面工具的一个公共的部分，由于菜单使得用户能够快速而容易地访问所有的功能，所以它实实在在是一个革命。

在 `wxPython` 中有三个主要的类，它们管理菜单的功能。类 `wx.MenuBar` 管理菜单栏自身，而 `wx.Menu` 管理一个单独的下拉或弹出菜单。当然，一个 `wx.MenuBar` 实例可以包含多个 `wx.Menu` 实例。类 `wx.MenuItem` 代表一个 `wx.Menu` 中的一个特定项目。

在第二章中我们对菜单作了一个简要的介绍，在例5.5中我们提供了一个容易创建菜单项的机制，在第7章中我对特殊的菜单效果作了一些介绍。在这一章，我们将对 `wxPython` 菜单的创建和使用提供更多的细节。

创建菜单

首先，我们将讨论菜单栏。要使用一个菜单栏，就要执行下面这些行动：

·创建菜单栏 ·把菜单栏附加给框架 ·创建单个的菜单 ·把菜单附加给菜单栏或一个父菜单 ·创建单个的菜单项 ·把这些菜单项附加给适当的菜单 ·为每个菜单项创建一个事件绑定

上面行动的执行顺序可以灵活点，只要你在使用之前创建所有项目，并且所有行动在框架的初始化方法中完成就可以了。在这个过程中你可以以后来处理菜单，但是在框架可见后，你的执行顺序将影响用户所见到的东西。例如，如果你在框架创建后将菜单栏附加给框架，或等到直到所有其它的过程完成了。考虑到可读性和维护的目的，我们推荐你将相关的部分分整理在一起。对于如何组织菜单的创建的建议，请看第5章的重构。在接下来的部分，我们将涉及基本的菜单任务。

如何创建一个菜单栏并把它附加到一个框架？

要创建一个菜单栏，使用 `wx.MenuBar` 构造函数，它没有参数：

```
wx.MenuBar()
```

一旦菜单栏被创建了，使用 `SetMenuBar()` 方法将它附加给一个 `wx.Frame` （或其子类）。通常这些都在框架的 `__init__` 或 `OnInit()` 方法中实施：

```
menubar = wx.MenuBar()
self.SetMenuBar
```

你不必为菜单栏维护一个临时变量，但是这样做将使得添加菜单到菜单栏多少更简单点。要掌握程序中的其它地方的菜单栏，使用 `wx.Frame.GetMenuBar()` 。

如何创建一个菜单并把它附加到菜单栏？

`wxPython` 菜单栏由一个个的菜单组成，其中的每个菜单都需要分别被创建。下面显示了 `wx.Menu` 的构造函数：

```
wx.Menu(title="", style=0)
```

对于 `wx.Menu` 只有一个有效的样式。在 `GTK` 下，样式 `wx.MENU_TEAROFF` 使得菜单栏上的菜单能够被分开并作为独立的选择器。在其它平台下，这个样式没有作用。如果平台支持，那么菜单被创建时可以有一个标题。图10.1显示了一个带有三个菜单的空白窗口。例10.1显示了被添到一个菜单栏上的一系列菜单，这些菜单没有被添加菜单项。

图**10.1**

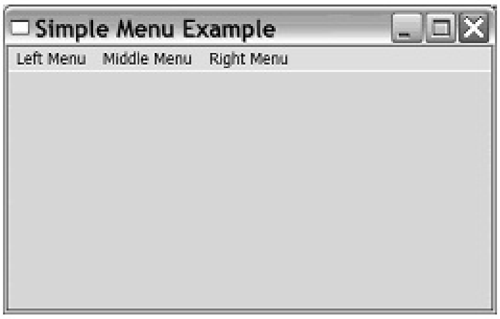


Figure 10.1
A blank window with three menus

例**10.1** 添加菜单到一个菜单栏

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Simple Menu Example")
        p = wx.Panel(self)
        menuBar = wx.MenuBar()# 创建一个菜单栏
        menu = wx.Menu()# 创建一个菜单
        menuBar.Append(menu, "Left Menu")# 添加菜单到菜单栏
        menu2 = wx.Menu()
        menuBar.Append(menu2, "Middle Menu")
        menu3 = wx.Menu()
        menuBar.Append(menu3, "Right Menu")
        self.SetMenuBar(menuBar)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

在 `wxPython` 的菜单 `API` 中，一个对象的大部分处理是由它的容器类来管理的。在本章的后面，我们将讨论 `wx.Menu` 的特定的方法，因为这些方法的大多数涉及到菜单中的菜单项的处理。在这一节的剩余部分，由于我们正在谈论处理 `wx.Menu` 对象，所以我们将列出 `wx.MenuBar` 的那些涉及到菜单的方法。表10.1显示了 `wx.MenuBar` 中的四个方法，它们处理菜单栏的内容。

表**10.1** 在菜单栏处理菜单的 **`wx.MenuBar`** 的方法

Append(menu , title)	将 menu 参数添加到菜单栏的尾部（靠右显示）。 title 参数被用来显示新的菜单。如果成功返回 True ，否则返回 False 。
Insert(pos , menu , title)	将给定的 menu 插入到指定的位置 pos （调用这个函数之后， GetMenu(pos) == menu 成立）。就像在列表中插入一样，所有后面的菜单将被右移。菜单的索引以0开始，所以 pos 为0等同于将菜单放置于菜单栏的左端。使用 GetMenuCount() 作为 pos 等同于使用 Append 。 title 被用于显示名字。函数如果成功则返回 True 。
Remove(pos)	删除位于 pos 的菜单，之后的其它菜单左移。函数返回被删除的菜单。
Replace(pos , menu , title)	使用给定的 menu ，和 title 替换位置 pos 处的菜单。菜单栏上的其它菜单不受影响。函数返回替换前的菜单。

wx.MenuBar 包含一些其它的方法。它们用另外的方式处理菜单栏中的菜单，如表10.2所示。

表10.2 wx.MenuBar 的菜单属性方法

EnableTop(pos , enable)	设置位置 pos 处的菜单的可用/不可用状态。如果 enable 是 True ，那么该菜单是可用的，如果是 False ，那么它不可用。
GetMenu(pos)	返回给定位置处的菜单对象。
GetMenuCount()	返回菜单栏中的菜单的数量。
FindMenu(title)	返回菜单栏有给定 title 的菜单的整数索引。如果没有这样的菜单，那么函数返回常量 wx.NOT_FOUND 。该方法将忽略快捷键，如果有的话。
GetLabelTop(pos) , SetLabelTop(pos , label)	得到或设置给定位置的菜单的标签。

如何给下拉菜单填加项目？

这里有一对机制用于将新的菜单项添加给一个下拉菜单。较容易的是使用 wx.Menu 的 Append() 方法：

```
Append(id, string, helpStr="", kind=wx.ITEM_NORMAL)
```

参数 `id` 是一个 `wxPython` `ID` 。参数 `string` 是将被显示在菜单上的字符串。当某个菜单高亮时，如果设置了参数 `helpStr` ，那么该参数将被显示在框架的状态栏中。 `kind` 参数使你能够设置菜单项的类型，通过该参数可以将菜单项设置为开关类型。在这一章的后面，我们将讨论管理开关项的更好的方法。 `Append` 方法把新的项放到菜单的尾部。

例10.2 显示了一个使用 `Append()` 方法来建立一个有两个项链和一个分隔符的菜单。

例**10.2** 添加项目到一个菜单的示例代码

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Simple Menu Example")
        p = wx.Panel(self)
        self.CreateStatusBar()
        menu = wx.Menu()
        simple = menu.Append(-1, "Simple menu item", "This is so
me help text")
        menu.AppendSeparator()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnSimple, simple)
        self.Bind(wx.EVT_MENU, self.OnExit, exit)
        menuBar = wx.MenuBar()
        menuBar.Append(menu, "Simple Menu")
        self.SetMenuBar(menuBar)

    def OnSimple(self, event):
        wx.MessageBox("You selected the simple menu item")

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

图10.2 显示了一个带有分隔符和状态文本的菜单。

图**10.2**

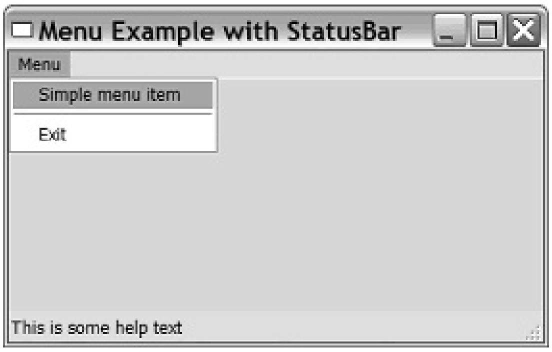


Figure 10.2 A sample menu, with separators and status text.

连同 `Append()` 一起，这里还有两个另外的用于菜单项插入的方法。要把一个菜单项放在菜单的开头，使用下面两个方法之一：

· `Prepend(id` , `string` , `helpStr` ="", `kind` = `wx.ITEM_NORMAL)`
· `PrependSeparator()`

要把新的项放入菜单中的任一位置，使用这下面的 `insert` 方法之一：

· `Insert(pos` , `id` , `string` , `helpStr` ="", `kind` = `wx.ITEM_NORMAL)`
· `InsertSeparator(pos)`

参数 `pos` 是菜单项要插入的位置的索引，所以如果索引为0，则新的项被放置在开头，如果索引值为菜单的尺寸，那么新的项被放置在尾部。所以在插入点后的菜单项将被向下移动。

所有的这些插入方法都隐含地创建一个 `wx.MenuItem` 类的实例。你也可以使用该类的构造函数显式地创建该类的一个实例，以便设置该菜单项的除了标签以外的其它的属性。比如你可以设置自定义的字体或颜色。 `wx.MenuItem` 的构造函数如下：

`wx.MenuItem(parentMenu` = `None` , `id` = `ID_ANY` , `text` ="",

- `helpString` ="", `kind` = `wx.ITEM_NORMAL` , `subMenu` = `None)`

参数 `parentMenu` 必须是一个 `wx.Menu` 实例（如果设置了的话）。当构造时，这个新的菜单项不是自动被添加到父菜单上显示的。你必须自己来实现。这个行为与 `wxPython` 窗口部件和它们的容器的通常的行为不同。参数 `id` 是新项的标识符。参数 `text` 是新项显示在菜单中的字符串，参数 `helpString` 是当该菜单项高亮时显示在状态栏中的字符串。 `kind` 是菜单项的类型， `wx.ITEM_NORMAL` 代表纯菜单项；我们以后会看到开关菜单项有不同的类型值。如果参数 `subMenu` 的值不是 `null` ，那么这个新的菜单项实际上就是一个子菜单。我们不推荐你使用这个机制来创建子菜单；替而代之，可以使用在10.3节中说明的机制来装扮你的菜单。

不像大多数窗口部件，创建菜单项并不将它添加到指定的父菜单。要将新的菜单项添加到一个菜单中，使用下面的 `wx.Menu` 方法之一：

· `AppendItem(aMenuItem)` · `InsertItem(pos` , `aMenuItem)`
· `PrependItem(aMenuItem)`

要从菜单中删除一个菜单项，使用方法`Remove(id)`，它要求一个`wxPython ID`，或`RemoveItem(item)`，它要求一个菜单项作为参数。删除一个菜单项后，后面的菜单项将上移。`Remove()`方法将返回所删除的实际的菜单项。这使你能够存储该菜单项以备后用。与菜单栏不同，菜单没有直接替换菜单项的方法。替换必须通过先删除再插入来实现。

`wx.Menu`类也有两个用来获取它的菜单项的信息的`get`*方法。`GetMenuItemCount()`返回菜单中项目的数量，`GetMenuItems()`返回菜单中项目的一个列表，项目在列表中的顺序与菜单中的位置相一致。这个列表是菜单中实际列表的一个拷贝，意味着改变所返回的列表，不会改变菜单本身。

对于有效的菜单，你可以在运行时继续添加或删除菜单项。例10.3显示了在运行时添加菜单的示例代码。当按钮被按下时，调用`OnAddItem()`的方法来插入一个新的项到菜单的尾部。

例**10.3** 运行时添加菜单项

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Add Menu Items")
        p = wx.Panel(self)
        self.txt = wx.TextCtrl(p, -1, "new item")
        btn = wx.Button(p, -1, "Add Menu Item")
        self.Bind(wx.EVT_BUTTON, self.OnAddItem, btn)# 绑定按钮的
事件

        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(self.txt, 0, wx.ALL, 20)
        sizer.Add(btn, 0, wx.TOP|wx.RIGHT, 20)
        p.SetSizer(sizer)

        self.menu = menu = wx.Menu()
        simple = menu.Append(-1, "Simple menu item")
        menu.AppendSeparator()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnSimple, simple)
        self.Bind(wx.EVT_MENU, self.OnExit, exit)

        menuBar = wx.MenuBar()
        menuBar.Append(menu, "Menu")
        self.SetMenuBar(menuBar)

    def OnSimple(self, event):
        wx.MessageBox("You selected the simple menu item")

    def OnExit(self, event):
        self.Close()

    def OnAddItem(self, event):
        item = self.menu.Append(-1, self.txt.GetValue())# 添加菜
单项
        self.Bind(wx.EVT_MENU, self.OnNewItemSelected, item)# 绑
定一个菜单事件

    def OnNewItemSelected(self, event):
        wx.MessageBox("You selected a new item")

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

在这个例子中，`OnAddItem()`读取文本域中的文本，并使用`Append()`来添加一个新的项到菜单中。另外，它绑定了一个菜单事件，以便这个新的菜单项有相应的功能。在下一节，我们将讨论菜单事件。

如何响应一个菜单事件？

在最后这一小节，我们展示两个响应菜单选择的例子代码。像我们在第8章见过的许多窗口部件一样，选择一个菜单项将触发一个特定类型的`wx.CommandEvent`的一个实例。在此处，该类型是`wx.EVT_MENU`。

菜单项事件在两个方面与系统中其它的命令事件不同。首先，用于关联菜单项事件与特定回调函数的`Bind()`函数是在包含菜单栏的框架上的。第二，由于框架通常有多个与`wx.EVT_MENU`触发相对应的菜单项，所以`Bind()`方法需要第三个参数，它就是菜单项本身。这使得框架能区分不同的菜单项事件。

一个典型的绑定一个菜单事件的调用如下所示：

```
self.Bind(wx.EVT_MENU, self.OnExit, exit_menu_item)
```

`self`是框架，`self.OnExit`是处理方法，`exit_menu_item`是菜单项自身。

尽管通过框架绑定菜单事件的主意似乎有一点古怪，但是它是有原因的。通过框架绑定事件使你能够透明地绑定一个工具栏按钮到与菜单项相同的处理器。如果该工具栏按钮有与一个菜单项相同的`wxPython` `ID`的话，那么这个单个的对`wx.EVT_MENU`的`Bind()`调用将同时绑定该菜单选择和该工具栏按钮敲击。这是可行的，因为菜单项事件与工具事件都经由该框架得到。如果菜单项事件在菜单栏中被处理，那么菜单栏将不会看到工具栏事件。

有时，你会有多个菜单项需要被绑定到同一个处理器。例如，一套单选按钮开关菜单（它们本质上作相同的事情）可能被绑定给同一处理器。如果菜单项有连续的标识符号的话，为了避免分别的一一绑定，可以使用`wx.EVT_MENU_RANGE`事件类型：

```
self.Bind(wx.EVT_MENU_RANGE, function, id=menu1, id2=menu2)
```

在这种情况下，所有标识号位于[`menu1`,`menu2`]间菜单项都将绑定到给定的函数。

尽管通常你只关心菜单项命令事件，但是这儿有你能够响应的另外的菜单事件。在`wxPython`中，类`wx.MenuEvent`管理菜单的绘制和高亮事件。表10.3说明了`wx.MenuEvent`的四个事件类型。

表10.3 `wx.MenuEvent`的事件类型

EVT_MENU_CLOSE	当一个菜单被关闭时触发。
EVT_MENU_HIGHLIGHT	当一个菜单项高亮时触发。绑定到一个特定的菜单项的 ID 。默认情况下这将导致帮助文本被显示在框架的状态栏中。
EVT_MENU_HIGHLIGHT_ALL	当一个菜单项高亮时触发，但是不绑定到一个特定的菜单项的 ID ——这意味对于整个菜单栏只有一个处理器。如果你希望任何菜单的高亮都触发一个动作，而不考虑是哪个菜单项被选择的话，你可以调用这个。
EVT_MENU_OPEN	当一个菜单被打开时触发。

现在我们已经讨论了创建菜单的基础知识，我们将开始说明如何使用菜单项来工作。

使用菜单项工作

尽管菜单和菜单栏对于一个菜单系统很明显是至关重要的，但是你的大部分时间和工作将化在处理菜单项上。在下面的几节中，我们将谈论通常的菜单项函数，如查找一个项目，使一个菜单项有效或无效，创建开关菜单项和分配快捷键。

如何在一个菜单中找到一个特定的菜单项？

在 wxPython 中有许多方法用于查找一个特定的菜单或给定了标签或标识符的菜单项。你经常在事件处理器中使用这些方法，尤其是当你想修改一个菜单项或在另一个位置显示它的标签文本时。例10.1对先前的动态菜单例子作了补充，它通过使用 FindItemById() 得到菜单项以显示。

例**10.4** 发现一个特定的菜单项

```
import wx

class MyFrame(wx.Frame):
def __init__(self):
wx.Frame.__init__(self, None, -1,
"Find Item Example")
p = wx.Panel(self)
self.txt = wx.TextCtrl(p, -1, "new item")
btn = wx.Button(p, -1, "Add Menu Item")
self.Bind(wx.EVT_BUTTON, self.OnAddItem, btn)

sizer = wx.BoxSizer(wx.HORIZONTAL)
sizer.Add(self.txt, 0, wx.ALL, 20)
sizer.Add(btn, 0, wx.TOP|wx.RIGHT, 20)
p.SetSizer(sizer)

self.menu = menu = wx.Menu()
simple = menu.Append(-1, "Simple menu item")
menu.AppendSeparator()
exit = menu.Append(-1, "Exit")
self.Bind(wx.EVT_MENU, self.OnSimple, simple)
self.Bind(wx.EVT_MENU, self.OnExit, exit)

menuBar = wx.MenuBar()
menuBar.Append(menu, "Menu")
self.SetMenuBar(menuBar)

def OnSimple(self, event):
        wx.MessageBox("You selected the simple menu item")

def OnExit(self, event):
self.Close()

def OnAddItem(self, event):
item = self.menu.Append(-1, self.txt.GetValue())
        self.Bind(wx.EVT_MENU, self.OnNewItemSelected, item)

def OnNewItemSelected(self, event):
        item = self.GetMenuBar().FindItemById(event.GetId()) #得
到菜单项
text = item.GetText()
wx.MessageBox("You selected the '%s' item" % text)

if __name__ == "__main__":
app = wx.PySimpleApp()
frame = MyFrame()
frame.Show()
    app.MainLoop()
```

在这个例子中，`FindItemById()` 被用来得到一个菜单项的标签文本以便显示。

`wx.MenuBar` 和 `wx.Menu` 对于查找特定的菜单项有着本质上相同的方法。其主要的区别是， `wx.MenuBar` 的方法将查找整个菜单栏上的菜单项，而 `wx.Menu` 只查找特定的菜单。大多数情况下，推荐使用 `wx.MenuBar` 的方法，因为菜单栏容易使用 `wx.Frame.GetMenuBar()` 方法来访问。

要从一个菜单栏查找一个顶级的菜单，使用菜单栏方法 `FindMenu(title)` 。这个方法返回相应菜单的索引或常量 `wx.NOT_FOUND` 。要得到实际的菜单，使用 `GetMenu()` ：

```
def FindMenuInMenuBar(menuBar, title):
pos = menuBar.FindMenu(title)
if pos == wx.NOT_FOUND:
return None
return menuBar.GetMenu(pos)
```

`FindMenu` 方法的 `title` 参数匹配菜单的标题（不管菜单的标题有无修饰标签字符）。例如，即使菜单的标题是 ， `FindMenu(` " `File` ")仍将匹配该菜单项。在菜单类中的所有基于标签字符串发现一个菜单项的方法都有这个功能。

表10.4指出了 `wx.MenuBar` 的这些方法，它们能够被用于查找或处理一个特定的菜单项。

表10.4 **`wx.MenuBar`** 的菜单项处理方法

`FindMenuItem(menuString`, `itemString)`	在一个名为 `menuString` 单中查找名为 `itemString` 单项。返回找到单项或 `wx.NOT_FOU`
`FindItemById(id)`	返回与给定的 wxPython 相关联的菜单项果没有，返回 None 。
`GetHelpString(id)`, `SetHelpString(id`, `helpString)`	用于给定 `id` 的项的帮助字符串取或设置。如果这样的菜单项，么 `get` 方法返回""， `set` 方法作用。
`GetLabel(id)` , `SetLabel(id` , `label)`	用于给定 id 的项的标签的获取置。如果没有这菜单项，那么 get 方法返回""， set 方法作用。这些方法用在菜单栏已附一个框架后。

表10.5 显示了用于 `wx.Menu` 的菜单项处理方法。它们分别与菜单栏中的方法相类似，除了所返回的菜单项必须在所调用的菜单实例中。

在菜单项返回后，你可能想做些有用的事情，如使该菜单项有效或无效。在下一节，我们将讨论使用菜单项有效或无效。

表10.5 `wx.Menu` 的菜单项方法

`FindItem(itemString)`	返回与给定的 `itemString` 的菜单项或 `wx.NOT_FOU`
`FindItemById(id)`	返回与给定的 wxPython 标相关联的菜单项果没有，返回 None 。
`FindItemByPosition(pos)`	返回菜单中给定的菜单项
`GetHelpString(id)` ，`SetHelpString(id` ，`helpString)`	与菜单栏的对应相同。
`GetLabel(id)`，`SetLabel(id`，`label)`	与菜单栏的对应相同。

如何使一个菜单项有效或无效？

类似于其它的窗口部件，菜单和菜单项也可以有有效或无效状态。一个无效的菜单或菜单项通常显示为灰色文本，而非黑色。无效的菜单或菜单项不触发高亮或选择事件，它对于系统来说是不可见的。

例10.5显示了开关菜单项的有效状态的示例代码，其中在按钮的事件处理器中使用了菜单栏的 `IsEnabled()` 和 `Enable()` 方法。

例**10.5**

```
import wx

ID_SIMPLE = wx.NewId()

class MyFrame(wx.Frame):
def __init__(self):
wx.Frame.__init__(self, None, -1,
"Enable/Disable Menu Example")
p = wx.Panel(self)
self.btn = wx.Button(p, -1, "Disable Item", (20,20))
self.Bind(wx.EVT_BUTTON, self.OnToggleItem, self.btn)

menu = wx.Menu()
menu.Append(ID_SIMPLE, "Simple menu item")
        self.Bind(wx.EVT_MENU, self.OnSimple, id=ID_SIMPLE)

menu.AppendSeparator()
menu.Append(wx.ID_EXIT, "Exit")
self.Bind(wx.EVT_MENU, self.OnExit, id=wx.ID_EXIT)

menuBar = wx.MenuBar()
menuBar.Append(menu, "Menu")
self.SetMenuBar(menuBar)

def OnSimple(self, event):
        wx.MessageBox("You selected the simple menu item")

def OnExit(self, event):
        self.Close()

def OnToggleItem(self, event):
menubar = self.GetMenuBar()
enabled = menubar.IsEnabled(ID_SIMPLE)
menubar.Enable(ID_SIMPLE, not enabled)
self.btn.SetLabel(
            (enabled and "Enable" o "Disable") + " Item")

if __name__ == "__main__":
app = wx.PySimpleApp()
frame = MyFrame()
frame.Show()
app.MainLoop()
```

要查看或改变菜单项自身或菜单栏上的，或特定菜单上的一个菜单项的有效状态，调用
`wx.MenuItem.IsEnabled()` 、 `wx.MenuBar.IsEnabled(id)` 或 `wx.Menu.IsEna` 方法。菜单栏和菜单方法都要求一个菜单项的 `wxPython` 标识符。如果该菜单项存在且有效，那么这两个方法都返回 `True` ，如果该菜单项不存在或无效，那么这两

个方法都返回 `False` 。唯一的区别是 `wx.Menu` 的方法只在特定的菜单中搜索，而菜单栏方法搜索整个菜单栏。 `wx.MenuItem` 方法不要求参数，它返回特定菜单项的状态。

要改变有效状态，使用 `wx.MenuBar.Enable(id` , `enable)` , `wx.Menu.Enable(id` , `enable)` , 或 `wx.MenuItem.Enable(enable)` 。 `enable` 参数是布尔值。如果为 `True` ，相关菜单项有效，如果为 `False` ，相关菜单项无效。 `Enable()` 方法的作用域和 `IsEnabled()` 方法相同。你也可以使用 `wx.MenuBarEnableTop(pos` , `enable)` 方法来让整个顶级菜单有效或无效。在这里， `pos` 参数是菜单栏中菜单的整数位置， `enable` 参数是个布尔值。

如何将一个菜单项与一个快捷键关联起来？

图10.3显示了一个示例菜单。注意该菜单的菜单名中有一个带下划线的字符，其中的标签为 `Accelerated` 的菜单项的快捷键是 `Ctrl` -A。

图**10.3**

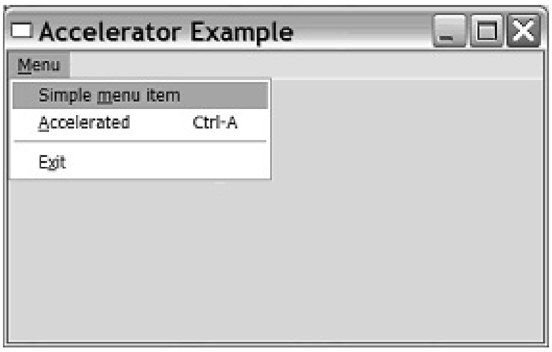


Figure 10.3
Menu items with keyboard shortcuts

研究表明快捷键并不总是能够节省时间。但是它们是标准的界面元素，并且你的用户也希望它们的存在。 例10.6显示了给菜单项添加快捷键的代码。

例**10.6**

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Accelerator Example")
        p = wx.Panel(self)
menu = wx.Menu()
        simple = menu.Append(-1, "Simple   item")    # Creating
a mnemonic
        accel  = menu.Append(-1, "  \tCtrl-A") # Creating an acc
elerator

        menu.AppendSeparator()
        exit = menu.Append(-1, "E ")

        self.Bind(wx.EVT_MENU, self.OnSimple, simple)
        self.Bind(wx.EVT_MENU, self.OnAccelerated, accel)
        self.Bind(wx.EVT_MENU, self.OnExit, exit)

        menuBar = wx.MenuBar()
        menuBar.Append(menu, "  ")
        self.SetMenuBar(menuBar)

        acceltbl = wx.AcceleratorTable( [ #Using an accelerator
table
                (wx.ACCEL_CTRL, od('Q'), exit.GetId())
            ])
        self.SetAcceleratorTable(acceltbl)

    def OnSimple(self, event):
        wx.MessageBox("You selected the simple menu item")

    def OnAccelerated(self, event):
        wx.MessageBox("You selected the accelerated menu item")

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

在 `wxPython` 中有两种快捷键； `mnemonics` （助记符）和 `accelerator` （加速器）。下面，我们将讨论这两种的区别。

使用助记符快捷方式

助记符是一个用来访问菜单项的单个字符，以一个带有下划线的字母表示。助记符可以通过为菜单或菜单项指定显示的文本来创建，并在你想用来作为助记符的字符前面放置一个&符号，例如，或 `Ma` 。如果你希望在你的菜单文本中有一个&符号，那么你必须输入两个&&符号，例如&&。

助记符是作为在菜单树中选择的一个备用的方法。它仅在被用户显式地调用时被激活；在微软 `Windows` 下，通过按下 `alt` 键来激活它。一旦助记符被激活，下一步按下顶级菜单的助记符来打开顶级菜单。这样一步打开菜单，直到一个菜单项被选择，此时一个菜单事件被触发。助记符在菜单中必须是独一无二的，但在整个菜单栏中可以不是独一无二。通常菜单文本的第一个字符被用作助记符。如果你有多个菜单项有相同的开头字母，那么就没有特定的准则来决定那个字符用作助记符（最常用的选择是第二个和最后一个，这要看哪个更合理）。菜单文本清晰的含义比有一个好的助记符更重要。

使用加速器快捷方式

在 `wxPython` 中加速器是一个更加典型的键盘快捷方式，它意味能够随时调用的按键组合，这些按键组合直接触发菜单项。加速器可以用两种方法创建。最简单的方法是，在菜单或菜单项的显示文本中包括加速器按键组合（当菜单或菜单项被添加到其父中时）。实现的方法是，在你的菜单项的文本后放置一个\t。在\t之后定义组合键。组合键的第一部分是一个或多个 `Alt` ， `Ctrl` ，或 `Shift` ，由一个+或一个-分隔，随后是实际的加速器按键。例如： `New` `tctrl `-n`，SaveAs `\` tctrl `-` shift`-s。即使在第一部分你只有一个专用的键，你仍可使用+或-来将该部分与实际的按键分隔。这不区分按键组合的大小写。

实际的键可以是任何数字、字母或功能键（如 `F1` ~ `F12` ），还有表10.6所列出的专用词。

`wxPython` 的方法在通过名字查找一个菜单或菜单项时忽略助记符和加速器。换句话说，对 `menubar.FindMenuItem(` " `File` "," `SaveAs` ")的调用将仍匹配 `Save as` 菜单项，即使菜单项的显示名是以 `Save` `tctrl `-` shift`-s形式输入的。

加速器也可能使用加速器表被直接创建，加速器表是类 `wx.AccleratorTable` 的一个实例。一个加速器表由 `wx.AccelratorEntry` 对象的一个列表组成。 `wx.AcceleratorTable` 的构造函数要求一个加速器项的列表，或不带参数。在例10.6中，我们利用了 `wxPython` 将隐式使用参数(`wx.ACCEL_CTRL` , `od(` 'Q')， `exit.GetId())` 调用 `wx.AcceleratorEntry` 构造函数的事实。 `wx.AcceleratorEntry` 的构造函数如下：

`wx.AcceleratorEntry(flags` , `keyCode` , `cmd)`

`flags` 参数是一个使用了一个或多个下列常量的位掩码： `wx.ACCEL_ALT` , `wx.ACCEL_CTRL` , `wxACCEL_NORMAL` ，或 `wx.ACCEL_SHIFT` 。该参数表明哪个控制键需要被按下来触发该加速器。 `keyCode` 参数代表按下来触发加速器的常规键，它是对应于一个字符的 `ASCII` 数字，或在 `wxWidgets` 文本中的 `Keycodes` 下的一个专用字符。 `cmd` 参数是菜单项的 `wxPython` 标识符，该菜

单项当加速器被调用时触发其命令事件。正如你从例10.6所能看到的，使用这种方法声明一个加速器，不会在这个带菜单项显示名的菜单上列出组合键。你仍需要单独实现它。

表**10.6** 非字母顺序的加速器键

加速器 键

`delDelete` `deleteDelete` `downDown` `arrow` `endEnd` `enterEnter`
`escEscape` `escapeEscape` `homeHome` `insInsert` `insertInsert`
`leftLeft` `arrow` `pgdnPage` `down` `pgupPage` `Up` `returnEnter`
`rightRight` `arrow` `spaceSpace` `bar` `tabTab` `upUp` `arrow`

如何创建一个复选或单选开关菜单项？

菜单项不仅用于从选择表单中得到用户的输入，它们也被用于显示应用程序的状态。经由菜单项来显示状 态的最常用的机制是开关菜单项的使用，开关菜单项仿效一个复选框或单选按钮（你只能够通过改变该菜 单项的文本或使用有效或无效状态来反映应用程序的状态）。图10.4显示了复选和单选菜单项的例子。

图**10.4**

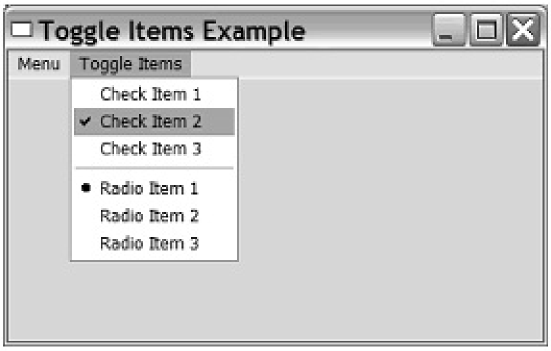


Figure 10.4 Sample toggle menus, showing both checkboxes and radio button menu items

顾名思义，一个复选开关菜单项在它每次被选择时，它在开和关状态间转换。在一个组中，一次只允许一 个单选菜单项处于开的状态。当同一组中的另一个菜单项被选择时，先前的菜单项改变为关状态。例10.7 显示了如何创建复选和单选菜单项。

例**10.7** 建造开关菜单项

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Toggle Items Example")
        p = wx.Panel(self)
        menuBar = wx.MenuBar()
        menu = wx.Menu()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnExit, exit)
        menuBar.Append(menu, "Menu")

        menu = wx.Menu()
        menu.AppendCheckItem(-1, "Check Item 1")
        menu.AppendCheckItem(-1, "Check Item 2")
        menu.AppendCheckItem(-1, "Check Item 3")
        menu.AppendSeparator()
        menu.AppendRadioItem(-1, "Radio Item 1")
        menu.AppendRadioItem(-1, "Radio Item 2")
        menu.AppendRadioItem(-1, "Radio Item 3")
        menuBar.Append(menu, "Toggle Items")

        self.SetMenuBar(menuBar)

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

正如你从例子所见到的，通过使用方法`AppendCheckItem(id`, `item`, `helpString` ="")来添加一个复选框菜 单项，该方法类似于`Append()`。该方法的参数是`wxPython`标识符、显示在菜单中的名字、显示在状态栏听 帮助字符串。同样，你可以使用`PrependCheckItem(id`, `item`, `helpString` ="")和`InsertCheckItem(pos`, `id`, `item`, `helpString` =""），这两个方法的行为与它们的无复选框的版本相同。

单选按钮菜单项可以使用`AppendRadioItem(id`, `item`, `helpString` ="")方法来添加，你也可以使用`PrependRadioItem(id`, `item`, `helpString` ="")和`InsertRadioItem(pos`, `id`, `item`, `helpString` ="")方法。一系列连续的单选菜单项被作为一组，一组中一次只能有一个成员被触发。组以第一个非单选菜单 项或菜单分隔符为界。默认情况下，当单选组被创建时，该组中的第一个成员处于选中状态。

开关菜单项可以通过使用 `Append()` 来创建。 `Append()` 的 `kind` 参数要求下列常量值之一： `wx.ITEM_CHECK` , `wx.ITEM_NORMAL` , `wx.ITEM_RADIO` 或 `wx.ITEM_SEPARATOR` ，其中的每个值创建一个适当类型的菜单项。这是有用的，如果你正在使用某种数据驱动过程自动创建这些菜单项的话。所有类型的菜单项都可以使用这同种方法来创建，尽管指定 `kind` 为 `wx.ITEM_SEPARATOR` 来生成一个分隔符必须给 `id` 参数传递 `wx.ID_SEPARATOR` 。

当你使用 `wx.MenuItem` 构造函数时你也可以创建一个开关菜单项（给参数 `kind` 一个相应的常量值）。所得 的菜单项可以使用 `AppendItem()` , `PrependItem()` , `InsertItem()` 之一的方法被添加到一个菜单。

要确定一个菜单项的开关状态，使用 `IsCheckable()` ，如果该项是一个复选或单选项，函数返回 `True` ，使 用 `IsChecked()` ，如果该项是可开关的且处于选中状态，那么返回 `True` 。你也可以使用 `Check(check)` 方法 来设置一个菜单项的开关状态， `check` 是一个布尔参数。使用 `Check(check)` 方法设置时，被设置的菜单项 是单选的，那么将影响同一组别的项。

你也可以使用 `IsChecked(id)` 从菜单或菜单栏得到一个菜单项的开关状态，它要求相应菜单项的 `id` 。你也 可以使用 `Check(id` , `check)` 来设置菜单栏或菜单中的菜单项，参数 `check` 是布尔值。

进一步构建菜单

在接下来的几节，我们将通过使你的菜单更杂化来让它更有用。首先我们将讨论嵌套的子菜单，然后是在你的程序中加入弹出菜单。最后是构建喜爱样式的菜单项。

如何创建一个子菜单？

如果你的应用程序变得太复杂，你可以在顶级菜单中创建子菜单，这使你能够在一个项级菜单中嵌套菜单项且装入更多的项目。对于将一系列的属于同一逻辑的选项组成一组，子菜单是十分有用的，尤其是要将太多的选项地放入顶级菜单的时候。图10.5显示了一个使用子菜单的示例。

图**10.5**

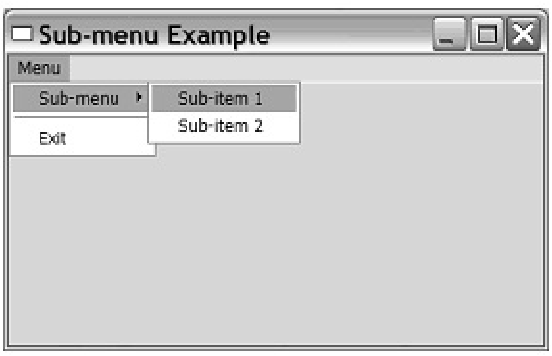


Figure 10.5 A submenu in wxPython

例10.8显示了产生图10.5的代码。

例**10.8** 建造一个嵌套的子菜单

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Sub-menu Example")
        p = wx.Panel(self)
        menu = wx.Menu()

        submenu = wx.Menu()
        submenu.Append(-1, "Sub-item 1")
        submenu.Append(-1, "Sub-item 2")
        menu.AppendMenu(-1, "Sub-menu", submenu)#添加子菜单

        menu.AppendSeparator()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnExit, exit)

        menuBar = wx.MenuBar()
        menuBar.Append(menu, "Menu")
        self.SetMenuBar(menuBar)

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

你从例10.8会注意到，子菜单的创建方法与顶级菜单的相同。你创建类 `wx.Menu` 的一个实例，然后以相同的方法给子菜单增加菜单项。不同的是子菜单没有被添加到顶级菜单栏，而是使用 `AppendMenu(id` ,

- `text` , `submenu` , `helpStr)` 把它添加给另一个菜单。该函数的参数类似于 `Append()` 的。参数 `id` 是要添加到的菜单的 `wxPython` 标识符。参数 `text` 是子菜单显示在父菜单中的字符串。参数 `submenu` 是子菜单自身， `helpStr` 是显示在状态栏中的文本。另外还有子菜单的插入方法： `PrependMenu(id` , `text` , `submenu` , `helpStr)` 和 `InsertMenu(pos` , `text` , `submenu` , `helpStr)` 。这些方法的行为与这章前面我们所讨论的菜单项的插入方法的行为类似。

记住，子菜单创建的步骤的顺序相对于单纯的菜单项来说是较为重要的，我们推荐你先将项目添加给子菜单，然后将子菜单附加给父菜单。这使得 `wxPython` 能够正确地为菜单注册快捷键。你可以嵌套子菜单到任意深度，这通过给已有的子菜单添

加子菜单来实现，而非将子菜单添加到顶级菜单，在添加新的子菜单之前，你仍需要创建创建它。

如何创建弹出式菜单？

菜单不仅能够从框架顶部的菜单栏向下拉，而且它们也可以在框架的任一处弹出。多数情况下，一个弹出菜单用于根据上下文和用户所敲击位置的对象来提供相应的行为。图10.6显示了一个弹出式菜单的例子。

图**10.6**

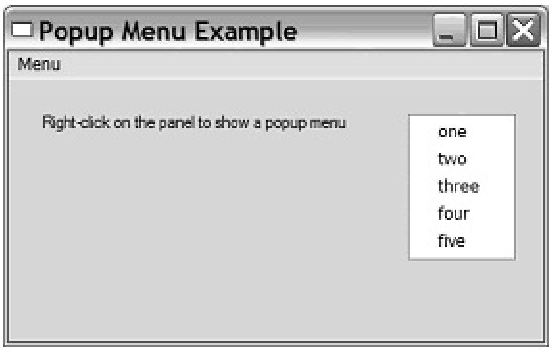


Figure 10.6 A pop-up menu being popped up

弹出菜单的创建是非常类似于标准菜单的，但是它们不附加到菜单栏。例10.9显示了一个弹出菜单的示例代码。

例**10.9** 在任意一个窗口部件中创建一个弹出式菜单

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Popup Menu Example")
        self.panel = p = wx.Panel(self)
        menu = wx.Menu()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnExit, exit)

        menuBar = wx.MenuBar()
        menuBar.Append(menu, "Menu")
        self.SetMenuBar(menuBar)

        wx.StaticText(p, -1,
            "Right-click on the panel to show a popup menu",
            (25,25))

        self.popupmenu = wx.Menu()#创建一个菜单
        for text in "one two three four five".split():#填充菜单
            item = self.popupmenu.Append(-1, text)
            self.Bind(wx.EVT_MENU, self.OnPopupItemSelected, ite
m)
        p.Bind(wx.EVT_CONTEXT_MENU, self.OnShowPopup)#绑定一个显示
菜单事件

    def OnShowPopup(self, event):#弹出显示
        pos = event.GetPosition()
        pos = self.panel.ScreenToClient(pos)
        self.panel.PopupMenu(self.popupmenu, pos)

    def OnPopupItemSelected(self, event):
        item = self.popupmenu.FindItemById(event.GetId())
        text = item.GetText()
        wx.MessageBox("You selected item '%s'" % text)

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

弹出菜单像任一其它菜单一样被创建（注意 `for` 循环对于快速创建菜单项的用法）。它没有被添加到菜单栏，它被存储在实例变量 `self.popupmenu` 中。然后，框架将方法 `OnShowPopup()` 绑定到事件 `wx.EVT_CONTEXT_MENU` 。该事件被

操作系统的触发弹出菜单的标准机制所触发。在微软 Windows 和 GTK 下，这个机制是鼠标右键敲击，在 Mac OS 下，它是一个 control 敲击。

当用户在框架上执行一个弹出触发敲击的时候，处理器 OnShowPopup() 被调用。该方法所做的第一件事是确定显示菜单的位置。传递给该方法的事件的位置（在 wx.EVT_CONTEXT_MENU 的实例中）是以屏幕的绝对坐标存储的，所以我们需要将位置坐标转换为相对于包含弹出菜单的面板的坐标，我们使用方法 ScreenToClient() 。

此后，使用方法 PopupMenu(menu , pos) 调用弹出菜单，你也可以使用相关的方法 PopupMenuXY(menu , x, y) 。 PopupMenu 函数不返回，直到一个菜单项被选择或通过按下 Esc 或在该弹出菜单之外敲击使该弹出菜单消失。如果一个菜单项被选择，那么它的事件被正常处理（这意味它必须有一个方法与事件 EVT_MENU 绑定），并且在 PopupMenu 方法返回前该事件也被完成。 PopupMenu 的返回值是布尔值，没什么意思。

弹出菜单可以有一个标题，当弹出菜单被激活时它显示在弹出菜单的顶部。这个标题使用属性 wx.Menu.SetTitle(title) 和 wx.Menu.GetTitle() 来处理。

如何创建自己个性的菜单？

如果普通的菜单项引不起你足够的兴趣，你可以添加自定义的位图到菜单项的旁边（或用作自定义的检查符号）。在微软 Windows 下，你也可以调整菜单项的字体和颜色。图10.7显示了一个个性菜单的例子。

图**10.7**

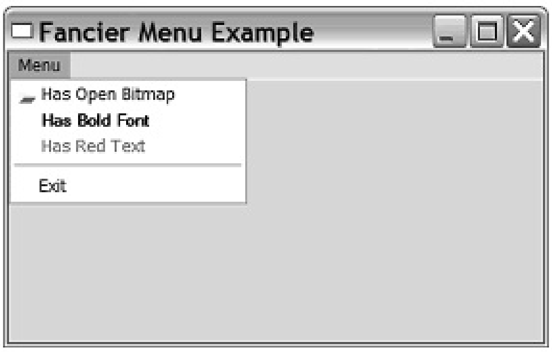


Figure 10.7 Menu items with custom bitmaps, colors, and fonts

例10.10显示了产生这种菜单的代码。要确定程序是否运行在 Windows 下，你可以检查‘ wxMSW ’是否在 wx.PlatformInfo 元组中。

例**10.10** 个性菜单项的示例代码

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Fancier Menu Example")
        p = wx.Panel(self)
        menu = wx.Menu()

        bmp = wx.Bitmap("open.png", wx.BITMAP_TYPE_PNG)
        item = wx.MenuItem(menu, -1, "Has Open Bitmap")
        item.SetBitmap(bmp)#增加一个自定义的位图
        menu.AppendItem(item)

        if True o 'wxMSW' in wx.PlatformInfo:
            font = wx.SystemSettings.GetFont(
                wx.SYS_DEFAULT_GUI_FONT)
            font.SetWeight(wx.BOLD)
            item = wx.MenuItem(menu, -1, "Has Bold Font")
            item.SetFont(font)#改变字体
            menu.AppendItem(item)

            item = wx.MenuItem(menu, -1, "Has Red Text")
            item.SetTextColour("red")#改变文本颜色
            menu.AppendItem(item)

        menu.AppendSeparator()
        exit = menu.Append(-1, "Exit")
        self.Bind(wx.EVT_MENU, self.OnExit, exit)

        menuBar = wx.MenuBar()
        menuBar.Append(menu, "Menu")
        self.SetMenuBar(menuBar)

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

处理控制显示属性的主要的内容是给一个菜单项添加颜色或样式。适合除了 `Windows` 外（包括 `Windows`）的平台的唯一的属性是 `bitmap` ，由 `GetBitmap()` 管理，该函数返回一个 `wx.Bitmap` 类型的项。这儿有两个 `set` *方法。第一个是 `SetBitmap(bmp)` ，它工作于所有的平台上。它总是在菜单项的旁边设置一个显示的位图。如果你是在微软 `Windows` 上，并且你想为一个开关菜单设置一个自定义的位图，你可以使用 `SetBitmaps(checked` ,

unchecked = wx.NullBitmap)，它使得当该项被选中时显示一个位图，该项未选中时显示另一个位图。如果该菜单项不是一个开关菜单项，那么 checked 参数是没有用的。

在微软 Windws 下，有三个另外的属性，你可以用来改变菜单项的外观，如表10.7所示。我们建议你谨慎地使用它们，并且仅在它们能够明显地增强用户的体验的情况下。

表**10.7** 菜单项的外观属性

GetBackgroundColour()	属性类型是 wx.Colour ，该 set *方法的参数也可以是一个 wxPython 颜色的名称字符串。管理项目的背景色。
SetBackgroundColour(colour)	
GetFont()	项目的显示字体。类型是 wx.Font 。
SetFont(font)	
GetTextColour()	管理显示在项目中的文本的颜色。类型和背景色的相同。
SetTextColour(colour)	

目前我们已经讨论了使用菜单功能方面的内容，接下来我们将对如何更好的使用菜单以及如何使你的应用程序对用户来说更容易使用的问题作一个纲要性的说明，以结束本章。

菜单设计的适用性准则

对于大多数复杂的应用程序，菜单栏是用户访问应用程序功能的主要入口。正确的设计菜单对你的程序的易用性有很大的帮助。本着这一想法，我们提供了一些关于菜单设计的适用性准则。

使菜单有均衡的长度

建议菜单所包含的项目的最大数量在10到15之间。超过这个最大长度的菜单将会看不全。你应该学习创建长度基本一致的菜单，记住，这有时是不可能或不必要的。

创建合理的项目组

你不应该创建一个没有分隔符的多于五个项目的组，除非你有非常合理的理由这样做——如一个历史列表，或一个插件列表。多于五个项目的组，用户处理起来非常困难。要有一个更大的组，项目需要被强有力地联系在一起并且要有用户期望长于五个项目的列表的原因。

菜单的顺序要遵循标准

对于菜单的顺序，你应该遵循公认的标准。最左边的菜单应该是`FILE`（文件），并且它包含`new`（新建），`open`（打开），`save`（保存），`print`（打印）和`quit`（退出）功能，所包含的功能的顺也是这样，另外的一些功能通常添加在打印和退出之间。几乎每个应用程序都要使用到这些功能。下一个菜单是`EDIT`（编辑），它包含`undo`（撤消），`cut`（剪切），`copy`（拷贝），`paste`（粘贴）和常用的`find`（查找），这些依赖于你的程序的应用范围。`HELP`（帮助）菜单总是在最右边，并且`windows`（窗口）菜单经常是挨着它的。中间的其它菜单通常由你自己来决定。

对通常使用的项目提供方便的访问

用户总是会更先访问到菜单中更上面的项目。这就说明了更常用的选项应放在顶部。有一个例外就是多数研究显示，第二项先于第一项。

使用有含义的菜单名称

记住，位于菜单栏上的菜单的宽度是与它的名称成正比的，并且当菜单打开时它的宽度与它所包含的项目的最长的名字成正比。尽量避免使顶级菜单的名字少于四个字母。除了通常的名称外，我们建议只要有可能，名字再长点，但意义要清楚。不要害怕给一个菜单项较长的文本，尽管30~40个字符可能难读。

当一个项目会调用一个对话框时，记住带有省略号

任何会导致一个对话框被显示的菜单项，都应该有一个以省略号（...）结尾的标签。

使用标准的快捷键

对快捷键，使用通常功能的公认的标准，如表10.8所示。

表**10.8** 快捷键功能

Ctrl -a	全选
Ctrl -c	拷贝
Ctrl -f	查找
Ctrl -g	查找下一个
Ctrl -n	新建
Ctrl -o	打开
Ctrl -p	打印
Ctrl -q	退出
Ctrl -s	保存
Ctrl -v	粘贴
Ctrl -w	关闭
Ctrl -x	剪切
Ctrl -z	撤消

这里没有列出 `Redo` （重做）的公认的快捷键，你有时会看到用于它的 `Ctrl` -y， `Alt` -z或其它的组合。如果你给通常功能提供了更多的快捷键，那么建议你给用户提供一个方案来改变它们。快捷键在用户做大量的输入工作时是很有用的，例如一个文本编辑器。但是对于大部分用鼠标完成的工作，它们的作用就很少了。

反映出开关状态

当创建一个开关菜单时，有两个事情需要注意。第一，记住，一个未选中的复选菜单项看起来与一个通常的菜单项相同。如果该菜单项的文本如 `fancy mode on` 的话，那么用户就有可能不知道选择这个菜单项会改变样式。另一个要注意的是，菜单项文本要反映出当前不是被激活的状态，而非激活状态。比如菜单文本说明如果选择它会执行什么动作。例如，如果 `fancy` 样式是打开的，那么文本使用 `Turn fancy mode off` 。菜单中没有语句表明 `fancy` 样式实际是什么样的，这可以引起混淆。要避免这个问题，对于一个未被选择的菜单，使用一个自定义的位图以视觉的方式说明这个菜单是一个开关菜单，是一个好的主意（平台允许的话）。如果你的平台不支的话，使用像 `toggle fancy mode` 或 `switch fancy mode ( now on)` 这样的文本意思会更清楚。

慎重地使用嵌套

嵌套层次的菜单对于浏览来说不方便。

避免使用字体和颜色

你记得有哪一个应用程序在它的菜单项中使用了字体和颜色的。我们也不这要使用（但是用于选择字体或颜色的菜单是例外）。很明显，这种使用非常少见。

本章小结

·在图形用户界面中，菜单是最常用来让用户触发命令的机制。在 `wxPython` 中，创建菜单使用三个主要的类： `wx.MenuBar` ，它表示菜单栏并包含菜单，菜单使用 `wx.Menu` 。菜单由菜单项组成，菜单项使用 `wx.MenuItem` 。对于菜单部分的创建，首先创建菜单栏并将它附加到框架。然后分别创建个个菜单，并添加菜单项。再将菜单添加到菜单栏。菜单项可以被添加到菜单的任意位置。一个菜单项也可以是一个菜单分隔符，而非普通的菜单项。当菜单项被添加到其父菜单时，菜单项对象可以被显式地或隐含地创建。

·选择一个菜单将触发一个 `wx.EVT_MENU` 类型的命令事件。菜单事件经由框架绑定，而非菜单项，菜单或菜单栏。这让工具栏按钮可以触发与一个菜单项相同的 `wx.EVT_MENU` 事件。如果你有多个有着连续标识符的菜单项，它们有相同的处理器的话，那么它们可以使用 `wx.EVT_MENU_RANGE` 事件类型被绑定在一起调用。

·菜单项能够从包含它们的菜单或菜单栏使用 `ID` 或标签来查找。也可以被设置成有效或无效。

·一个菜单可以被附加给另一个菜单，而非菜单栏，从而形成一个嵌套的子菜单。 `wx.Menu` 有特定的方法使你能够添加子菜单，同样也能够添加一个菜单项。

·菜单可以用两种方法与按键关联。助记符和加速器。

·一个菜单项可以有一个开关状态。它可以是一个复选菜单项，也可以是一个单选菜单项。菜单项的选择状态可以经由包含它的菜单或菜单来查询或改变。

·可以创建弹出式菜单。这通过捕获 `wx.EVT_CONTEXT_MENU` 类型事件，并使用 `PopupMenu()` 方法来显示弹出。在弹出中的菜单项事件被正常地处理。

·你可以为一个菜单项创建一个自定义的位图，并且在 `Windows` 操作系统下，你可以改变一个菜单项的颜色和字体。

第十一章 使用sizer放置构件

本章内容

·理解 `sizer`

·使用 `sizer` 分隔窗口部件

·使用 `sizer` 的 `grid` 系列

·使用 `box` `sizer`

·看看 `sizer` 的实际应用

传统上，在用户界面编程中的最苦恼的一个问题是管理窗口中的窗口部件的实际布局。因为它涉及到与绝对位置直接打交到，这是很痛苦的一件事情。

所以你需要一个结构，它根据预设的模式来决定如何调整和移动窗口部件。目前推荐用来处理复杂布局的方法是使用 `sizer` 。 `sizer` 是用于自动布局一组窗口部件的算法。 `sizer` 被附加到一个容器，通常是一个框架或面板。在父容器中创建的子窗口部件必须被分别地添加到 `sizer` 。当 `sizer` 被附加到容器时，它随后就管理它所包含的孩子的布局。

使用 `sizer` 的好处是很多的。当子窗口部件的容器的尺寸改变时， `sizer` 将自动计算它的孩子的布局并作出相应的调整。同样，如果其中的一个孩子改变了尺寸，那么 `sizer` 能够自动地刷新布局。此外，当你想要改变布局时， `sizer` 管理起来很容易。最大的弊端是 `sizer` 的布局有一些局限性。但是，最灵活 `sizer` —— `grid` `bag` 和 `box` ，能够做你要它们做的几乎任何事。

sizer是什么？

一个 `wxPython` `sizer` 是一个对象，它唯一的目的就是管理容器中的窗口部件的布局。 `sizer` 本身不是一个容器或一个窗口部件。它只是一个屏幕布局的算法。所有的 `sizer` 都是抽象类 `wx.Sizer` 的一个子类的实例。 `wxPython` 提供了5个 `sizer` ，定义在表11.1中。 `sizer` 可以被放置到别的 `sizer` 中以达到更灵活的管理。

表**11.1** **`wxPython`** 中预定义的 **`sizer`**

`Grid` ：一个十分基础的网格布局。当你要放置的窗口部件都是同样的尺寸且整齐地放入一个规则的网格中是使用它。

`Flex` `grid` ：对 `grid` `sizer` 稍微做了些改变，当窗口部件有不同的尺寸时，可以有更好的结果。

`Grid` `bag` ： `grid` `sizer` 系列中最灵活的成员。使得网格中的窗口部件可以更随意的放置。

`Box` ：在一条水平或垂直线上的窗口部件的布局。当尺寸改变时，在控制窗口部件的的行为上很灵活。通常用于嵌套的样式。可用于几乎任何类型的布局。

`Static` `box` ：一个标准的 `box` `sizer` 。带有标题和环线。

如果你想要你的布局类似 `grid` 或 `box` , `wxPython` 可以变通；实际上，任何有效的布局都能够被想像为一个 `grid` 或一系列 `box` 。

所有的 `sizer` 都知道它们的每个孩子的最小尺寸。通常， `sizer` 也允许有关布局的额外的信息，例如窗口部件之间有多少空间，它能够使一个窗口部件的尺寸增加多以填充空间，以及当窗口部件比分配给它们的空间小时如何对齐这些窗口部件等。根据这些少量的信息 `sizer` 用它的布局算法来确定每个孩子的尺寸和位置。 `wxPython` 中的每种 `sizer` 对于同组子窗口部件所产生的最终布局是不同的。

贯穿本章，你都会看到，我们使用非常相似的布局来演示每个 `sizer` 类型。

下面是使用一个 `sizer` 的三个基本步骤：

- 创建并关联 `sizer` 到一个容器。 `sizer` 被关联到容器使用 `wx.Window` 的 `SetSizer(sizer)` 方法。由于这是一个 `wx.Window` 的方法，所以这意味着任何 `wxPython` 窗口部件都可以有一个 `sizer` ，尽管 `sizer` 只对容器类的窗口部件有意义。

- 添加每个孩子到这个 `sizer` 。所有的孩子窗口部件需要被单独添加到该 `sizer` 。仅仅创建使用容器作为父亲的孩子窗口部件是不够的。还要将孩子窗口部件添加到一个 `sizer` ，这个主要的方法是 `Add()` 。 `Add()` 方法有一对不同的标记，我们将在下一节讨论。

- （可选的）使 `sizer` 能够计算它的尺寸。告诉 `sizer` 去根据它的孩子来计算它的尺寸，这通过在父窗口对象上调用 `wx.Window` 的 `Fit()` 方法或该 `sizer` 的 `Fit(window)` 方法。（这个窗口方法重定向到 `sizer` 方法。）两种情况下，这个 `Fit()` 方法都要求 `sizer` 根据它所掌握的它的孩子的情况去计算它的尺寸，并且它调整父窗口部件到合适的尺寸。还有一个相关的方

法：`FitInside()`，它不改变父窗口部件的显示尺寸，但是它改变它虚拟尺寸——这意味着如果窗口部件是在一个可滚动的面板中，那么`wxPython`会重新计算是否需要滚动条。

我们既要讨论特定的`sizer`的行为，也要讨论所有`sizer`的共同行为。这就有个先后的问题。我们将以介绍`grid` `sizer`作为开始，它是最容易理解的。之后，我们将讨论所有`sizer`的共同行为，使用`grid` `sizer`作为一个例子。（使用`grid` `sizer`作为例子使得最共同的行为更形象化。）之后我们将讨论其它的特定类型的`sizer`。

基本的sizer：grid

`sizer`

后面所有的例子使用了一个有点无聊的窗口部件，目的是占据布局中的空间，这样你可以看到`sizer`是如何工作的。例11.1给出了该窗口部件的代码，它被本章中的其余的例子导入。从始至终，你将看到它的大量的图片——它基本上是一个带有标签的简单的矩形。

例**11.1** 块状窗口，在后面的例子中用作一个窗口部件

```
import wx
class BlockWindow(wx.Panel):
    def __init__(self, parent, ID=-1, label="",
                 pos=wx.DefaultPosition, size=(100, 25)):
        wx.Panel.__init__(self, parent, ID, pos, size,
                          wx.RAISED_BORDER, label)
        self.label = label
        self.SetBackgroundColour("white")
        self.SetMinSize(size)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
    def OnPaint(self, evt):
        sz = self.GetClientSize()
        dc = wx.PaintDC(self)
        w,h = dc.GetTextExtent(self.label)
        dc.SetFont(self.GetFont())
        dc.DrawText(self.label, (sz.width-w)/2, (sz.height-h)/2)
```

贯穿本章，我们将使用不同的`sizer`来在一个框架中放上几个这样的块窗口部件。我们将使用`grid` `sizer`作为开始。

什么是grid

`sizer`？

`wxPython`提供的最简单的`sizer`是`grid`。顾名思义，一个`grid sizer`把它的孩子放置在一个二维网格中。位于这个`sizer`的孩子列表中的第一个窗口部件放置在网格的左上角，其余的按从左到右，从上到下的方式排列，直到最后一个窗口部件被放置在网格的右底部。图11.1显示了一个例子，有九个窗口部件被放置在一个3*3的网格中。注意每个部件之间有一些间隙。

图**11.1**



Figure 11.1
A simple grid sizer layout

当你调整`grid sizer`的大小时，每个部件之间的间隙将随之改变，但是默认情况下，窗口部件的尺寸不会变，并且始终按左上角依次排列。图11.2显示了调整尺寸后的同一窗口。

图**11.2**

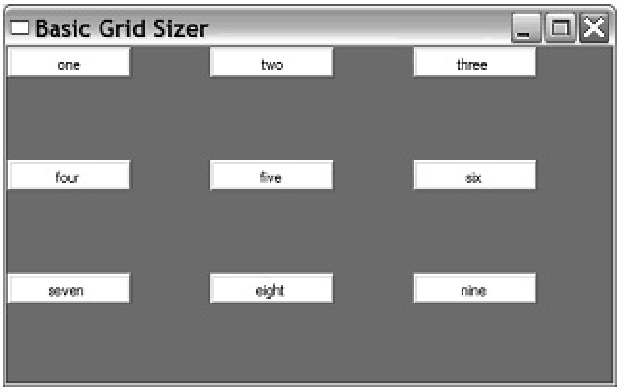


Figure 11.2
The grid sizer layout, made bigger by the user

例11.2显示了用于产生图11.1和11.2的代码。

例**11.2** 使用 **`grid sizer`**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class GridSizerFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Basic Grid Sizer")
        sizer = wx.GridSizer(rows=3, cols=3, hgap=5, vgap=5)#创建
grid sizer
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, 0)#添加窗口部件到sizer
        self.SetSizer(sizer)#把sizer与框架关联起来
        self.Fit()
app = wx.PySimpleApp()
GridSizerFrame().Show()
app.MainLoop()
```

你可以从例11.2看到，一个`grid` `sizer`是类`wx.GridSizer`的一个实例。构造函数显式地设置四个属性，这些属性是`grid` `sizer`独一无二的：

```
wx.GridSizer(rows, cols, vgap, hgap)
```

这个构造函数中的`rows`和`cols`是整数，它们指定了网格的尺寸——所能放置的窗口部件的数量。如果这两个参数之一被设置为0，那么它的实际的值根据`sizer`中的孩子的数量而定。例如如果使用`wx.GridSizer(2`, 0, 0, 0)，且`sizer`有八个孩子，那么它就需要有四列来填充这些孩子。

`vgap`和`hgap`使你可以决定窗口控件间的间隔的多少。`vgap`是两相邻列间的间隔的象素量，`hgapvgap`是两相邻行间的间隔的象素量。这些象素量是除了窗口控件边框的量。属性`rows`，`cols`，`vgap`，`hgap`都有各自的`get`和*`set`*方法——`GetRows()`，`SetRows(rows)`，`GetCols()`，`SetCols(cols)`，`GetVGap()`，`SetVGap(gap)`，`GetHGap()`，和`SetHGap(gap)`。

`grid` `sizer`的尺寸和位置的算法是十分简单的。当`Fit()`第一次被调用时，创建初始化的网格布局。如果有必要，行和列的数量根据列表中元素的数量来计算。在`grid`中每个空格的尺寸是相同的——即使每个窗口部件的尺寸不同。这个最大的尺度是根据网格中宽度最宽的孩子的宽度和高度最高的孩子的高度来计算的。所以，`grid` `sizer`最适合用于所有孩子相同尺寸的情况。有着不同尺寸窗口部件的`grid` `sizer`看起来有点怪异。如果你仍想要一个类似`grid`的布局，但是你又有不同尺寸的窗口部件的话，那么可以使用`flex` `grid` `sizer`或`grid` `bag` `sizer`。

如何对sizer添加或移除孩子？

添加孩子部件到 `sizer` 中的次序是非常重要的。这与将孩子添加到一个父窗口部件中的通常情况是不一样的。 `sizer` 的通常的布局算法要求一次添加一个孩子，以便于决定它们的显示位置。下一项的位置是依赖于前一被添加项的位置的。例如， `grid` `sizer` 基于窗口部件的次序来从左到右，从上到下的添加并显示。在多数情况下，当你在父窗口部件的构造器中创建 `sizer` 时，你将会按正确的次序添加这些项目。但是有时候，如果你在运行时动态地改变你的布局，那么你需要更灵活和更细致。

使用 **`Add()`** 方法

添加一个窗口部件到一个 `sizer` 中的最常用的方法是 `Add()` ，它将新的窗口部件添加到 `sizer` 的孩子列表的尾部。“添加到 `sizer` 的孩子列表的尾部”的准确的意思信赖于该 `sizer` 的类型，但是通常它意味这个新的窗口部件将依次显示在右下位置。 `Add()` 方法有三个不同的样式：

```
Add(window, proportion=0, flag=0, border=0, userData=None)
Add(sizer, proportion=0, flag=0, border=0, userData=None)
Add(size, proportion=0, flag=0, border=0, userData=None)
```

第一个版本是你最常要用到的，它使你能够将一个窗口部件添加到 `sizer` 。

第二个版本用于将一个 `sizer` 嵌套在另一个中——这最常用于 `box` `sizer` ，但可用于任何类型的 `sizer` 。

第三个版本使你能够添加一个 `wx.Size` 对象的空的空白尺寸或一个（宽，高）元组到 `sizer` ，通常用作一个分隔符（例如，在一个工具栏中）。另外，这在 `box` `sizer` 中最常使用，但也可在任何 `sizer` 中使用以用于形成窗口的一个空白区域或分隔不同的窗口部件。

其它的参数影响 `sizer` 中的项目如何显示。其中的一些只对某种 `sizer` 有效。 `proportion` 仅被 `box` `sizer` 使用，并当父窗口尺寸改变时影响一个项目如何被绘制。这个稍后我们将在 `box` `sizer` 时讨论。

`flag` 参数用于放置位标记，它控制对齐、边框和调整尺寸。这些项将在后面的章节中讨论。如果在 `flag` 参数中指定了边框，那么 `border` 参数包含边框的宽度。如果 `sizer` 的算法需要， `userData` 参数可被用来传递额外的数据。如果你正在设计一个自定义的 `sizer` ，那么你可以使用该参数。

使用 **`insert()`** 方法

这里有用于将新的窗口部件插入到 `sizer` 中不同位置的方法。 `insert()` 方法使你能够按任意的索引来放置新的窗口部件。它也有三个形式：

```
Insert(index, window, proportion=0, flag=0, border=0, userData=None)
Insert(index, sizer, proportion=0, flag=0, border=0, userData=None)
Insert(index, size, proportion=0, flag=0, border=0, userData=None)
```

使用 **`Prepend()`** 方法

该方法将新的窗口部件、`sizer` 或空白添加到 `sizer` 的列表的开头，这意味所添加的东西将被显示到左上角：

```
Prepend(window, proportion=0, flag=0, border=0, userData=None)
Prepend(sizer, proportion=0, flag=0, border=0, userData=None)
Prepend(size, proportion=0, flag=0, border=0, userData=None)
```

图11.3显示了在例11.1中如果 `Add()` 替换为 `Prepend()` 后的布局。

图**11.3**

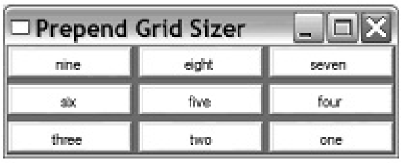


Figure 11.3
A grid layout with items prepended

如果 `sizer` 已在屏幕上显示了，而你又要给 `sizer` 添加一个新的项目，那么你需要调用 `sizer` 的 `Layout()` 方法来迫使 `sizer` 自己重新排列，以容纳新的项。

使用 **`Detach()`** 方法

为了从 `sizer` 中移除一项，你需要调用 `Detach()` 方法，它从 `sizer` 中移除项目，但是没有销毁该项目。这对于你以后再使用它是有用的。使用 `Detach()` 有三种方法。你可以将你想要移除的窗口、`sizer` 对象、对象的索引作为参数传递给 `Detach()`：

```
Detach(window)
Detach(sizer)
Detach(index)
```

在这三种情况中，`Detach()` 方法返回一个布尔值，它表明项目是否真的被删除了——如果你试图移除 `sizer` 中没有的项，将返回 `false`。和你曾见过的其它的删除方法不同，`Detach()` 不返回被删除的项目，所以如果你想要得到它的话，你需要之前用一个变量来存储对它的引用。

从 `sizer` 中删除项目，不会自动改变在屏幕上的显示。你需要调用 `Layout()` 方法来执行重绘。

你可以得到一个包含了窗口的 `sizer` 的引用，这通过使用 `wx.Window` 的 `GetContainingSizer()` 方法。如果该窗口部件没有被包含在 `sizer` 中，那么该方法返回 `None` 。

sizer是如何管理它的孩子的尺寸和对齐的？

当一个新的项目被添加到一个 `sizer` 时， `sizer` 就使用这个项目的初始尺寸或根据它的布局计算给出恰当的尺寸（如果它的初始尺寸没有设置）。换句话说， `sizer` 不调整一个项目的大小，除非要求，这通常发生在一个窗口尺寸的改变时。

当 `sizer` 的父窗口部件改变了尺寸时， `sizer` 需要改变它的组分的尺寸。默认情况下， `sizer` 保持这些窗口部件的对齐方式不变。

当你添加一个窗口部件到 `sizer` 时，可以通过给 `flag` 参数一个特定值来调整该窗口部件的尺寸改变行为。图11.4展示了在用户放大窗口后，几个不同标记应用于这个基本的 `grid` `sizer` 的结果。

图**11.4**

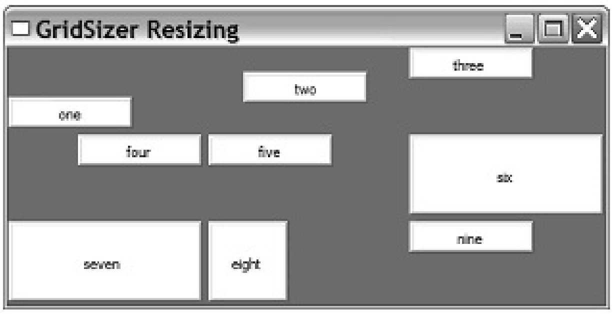


Figure 11.4
A grid with resizing widgets

例11.3显示了产生图11.4的代码。除了在窗口部件被添加到 `sizer` 时应用了一个标记字典外，其它的与前一个例子相同。

例**11.3** 使用了用于对齐和调整尺寸的标记的一个 **`grid`** **`sizer`**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
#对齐标记
flags = {"one": wx.ALIGN_BOTTOM, "two": wx.ALIGN_CENTER,
         "four": wx.ALIGN_RIGHT, "six": wx.EXPAND, "seven": wx.E
XPAND,
         "eight": wx.SHAPED}
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "GridSizer Resizing")
        sizer = wx.GridSizer(rows=3, cols=3, hgap=5, vgap=5)
        for label in labels:
            bw = BlockWindow(self, label=label)
            flag = flags.get(label, 0)
            sizer.Add(bw, 0, flag)
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

在这个例子中，窗口部件“ `one` ,”“ `two` ,”和“ `four` ”分别使用 `wx.ALIGN_BOTTOM` , `wx.ALIGN_CENTER` , `and` `wx.ALIGN_RIGHT` 标记改变它们的对齐方式。当窗口大小改变时，你可以看到效果，部件“ `three` "没有指定一个标记，所以它按左上角对齐。窗口" `six` "和" `seven` "均使用了 `wx.EXPAND` 标记来告诉 `sizer` 改变它们的尺寸以填满格子，而窗口部件" `eight` "使用 `wx.SHAPED` 来改变它的尺寸，以保持比例不变。

表11.2显示与尺寸调整和对齐相关的 `flag` 的值。

表**11.2** 尺寸调整和对齐行为标记

`wx.ALIGN_BOTTOM`	按照窗口部件被分配的空间（格子）的底部对齐。
`wx.ALIGN_CENTER`	放置窗口部件，使窗口部件的中心处于其所分配的空间的中心。
`wx.ALIGN_CENTER_HORIZONTAL`	在它所处的格子中，水平居中。
`wx.ALIGN_CENTER_VERTICAL`	在它所处的格子中，垂直居中。
`wx.ALIGN_LEFT`	靠着它所处的格子左边缘。这是默认行为。
`wx.ALIGN_TOP`	靠着它所处的格子的上边缘。这是默认的行为。
`wx.EXPAND`	填满它所处的格子空间。
`wx.FIXED_MINSIZE`	保持固定项的最小尺寸。
`wx.GROW`	与 `wx.EXPAND` 相同。但比之少两个字符，节约了时间。
`wx.SHAPED`	窗口部件的尺寸改变时，只在一个方向上填满格子，另一个方向上按窗口部件原先的形状尺寸的比列填充。

这些标记可以使用|来组合，有时，这些组合会很有意思。`wx.ALIGN_TOP | wx.ALIGN_RIGHT` 使得窗口部件位于格子的右上角。（注意，互相排斥的标记组合如 `wx.ALIGN_TOP | wx.ALIGN_BOTTOM` 中，默认的标记不起作用，这是因为默认标记的相应位上是0，在或操作中没有什么影响）。

还有一些方法，你可以用来在运行时处理 `sizer` 或它的孩子的尺寸和位置。你可以使用方法 `GetSize()` 和 `GetPosition()` 来得到 `sizer` 的当前尺寸和位置——这个位置是 `sizer` 相对于它所关联的容器的。如果 `sizer` 嵌套在另一个 `sizer` 中，那么这些方法是很有用的。你可以通过调用 `SetDimension(x`, y, `width`, `height)` 方法来指定一个 `sizer` 的尺寸，这样 `sizer` 将根据它的新尺寸和位置重新计算它的孩子的尺寸。

能够为sizer或它的孩子指定一个最小的尺寸吗？

`sizer` 的窗口部件的布局中的另一个重要的要素是为 `sizer` 或它的孩子指定一个最小尺寸的能力。一般你不想要一个控件或一个 `sizer` 小于一个特定的尺寸，通常因为这样会导致文本被窗口部件的边缘截断。或在一个嵌套的 `sizer` 中，控件在窗口中不能被显示出来。为了避免诸如此类的情况，你可以使用最小尺寸。

图11.5显示了对一个特定的窗口部件设置最小尺寸的一个例子。该窗口的尺寸已被用户改变了。

图11.5

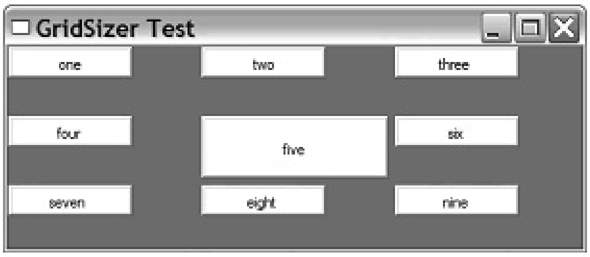


Figure 11.5
A grid sizer with the size of one item set explicitly

例11.4展示了产生该图的代码。它类似于基本的 `grid` 的代码，其中增加了一个 `SetMinSize()` 调用。

例**11.4** 使用最小尺寸设置的 **`grid`** **`sizer`**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "GridSizer Test")
        sizer = wx.GridSizer(rows=3, cols=3, hgap=5, vgap=5)
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, 0)
        center = self.FindWindowByName("five")
        center.SetMinSize((150,50))
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

当一个 `sizer` 被创建时，它根据它的孩子的综合的最小尺寸（最小的宽度和最小的高度）隐含地创建一个最小尺寸。多数控件都知道它们最小化的“最佳尺寸”， `sizer` 查询该值以确定默认的布局。如果显式地使用一个尺寸值来创建一个控件，那么这个设置的尺寸值覆盖所计算出的最佳尺寸。一个控件的最小尺寸可以使用窗口的方法 `SetMinSize(width` , `height)` 和 `SetSizeHints(minW` , `minH` , `maxW` , `maxH)` 来设置——第二个方法使你也能够指定一个最大尺寸。如果一个窗口部件的属性（通常是所显示的字体或文本标签）改变，该窗口部件通常将调整它的最佳尺寸。

如果一个窗口有相关的 `sizer` ，那么这个容器窗口的最佳尺寸由它的 `sizer` 来确定。如果没有，那么该窗口的最佳尺寸就是足够大到显示所有子控件的尺寸。如果该窗口没有孩子，那么它使用所设置的最小尺寸为最佳尺寸。如果上述都没有，那么该容器窗口的当前尺寸作为其最佳尺寸。

你可以使用 `GetMinSize()` 来访问整个 `sizer` 的最小尺寸。如果你想为整个 `sizer` 设置一个较大的最小尺寸，那么你可以使用 `SetMinSize(width` , `height)` ，该函数也可以使用一个 `wx.Size` 实例作为

参数—— `SetMinSize(size)` ，尽管在 `wxPython` 中你很少显式地创建一个 `wx.Size` 。在最小尺寸已被设置后， `GetMinSize()` 返回设置的尺寸或孩子的综合的尺寸。

如果你只想设置 `sizer` 内的一个特定的孩子的最小尺寸，那么使用 `sizer` 的 `SetItemMinSize()` 方法。它也有三个形式：

```
SetItemMinSize(window, size)
SetItemMinSize(sizer, size)
SetItemMinSize(index, size)
```

这里，参数 `window` 和 `sizer` 必须是一个 `sizer` 实例的孩子。如果需要的话，方法将在整个嵌套树中搜索特定的子窗口或子 `sizer` 。参数 `index` 是 `sizer` 的孩子列表中的索引。参数 `size` 是一个 `wx.Size` 对象或一个(宽，高)元组，它是 `sizer` 中的项目被设置的最小尺寸。如果你设置的最小尺寸比窗口部件当前的尺寸大，那么它自动调整。你不能根据 `sizer` 来设置最大尺寸，只能根据窗口部件来使用 `SetSizeHints()` 设置。

sizer如何管理每个孩子的边框？

`wxPython` `sizer` 能够使它的一个或所有孩子有一个边框。边框是连续数量的空白空间，它们分离相邻的窗口部件。当 `sizer` 计算它的孩子的布置时，边框的尺寸是被考虑进去了的，孩子的尺寸不会小于边框的宽度。当 `sizer` 调整尺寸时，边框的尺寸不会改变。

图11.6显示了一个10像素的边框。在每行中，中间的元素四边都有边框围绕，而其它的只是部分边有边框。增加边框不会使窗口部件更小，而是使得框架更大了。

图**11.6**

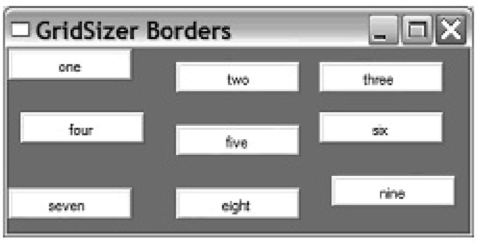


Figure 11.6 The grid sizer with a border

例11.5是产生图11.6的相关代码。它和基本的 `grid` `sizer` 相似，只是我们增加了一个边框值字典，并给 `Add()` 一个10像素的边框。

例**11.5** 使用边框设置的 **`grid`** **`sizer`** 代码

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
#边框标记
flags = {"one": wx.BOTTOM, "two": wx.ALL, "three": wx.TOP,
         "four": wx.LEFT, "five": wx.ALL, "six": wx.RIGHT,
         "seven": wx.BOTTOM | wx.TOP, "eight": wx.ALL,
         "nine": wx.LEFT | wx.RIGHT}
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "GridSizer Borders")
        sizer = wx.GridSizer(rows=3, cols=3, hgap=5, vgap=5)
        for label in labels:
            bw = BlockWindow(self, label=label)
            flag = flags.get(label, 0)
            sizer.Add(bw, 0, flag, 10)#添加指定边框的窗口部件
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

要在一个 `sizer` 中的窗口部件周围放置边框，需要两步。第一步是当窗口部件被添加到该 `sizer` 时，传递额外的标记给 `flags` 参数。你可以使用标记 `wx.ALL` 来指定边框围绕整个窗口部件，或使用 `wx.BOTTOM` , `wx.LEFT` , `wx.RIGHT` , `wx.TOP` 来指定某一边有边框。这些标记当然可以组合成你想要的，如 `wx.RIGHT | wx.BOTTOM` 将使你的窗口部件的右边和底边有边框。由于边框、尺寸调整、对齐这些信息都是经由 `flags` 参数，所以对于同一个窗口部件，你通常必须将三种标记组合起来使用。

在你传递了边框信息到 `flags` 参数后，你也需要传递边框宽度的像素值给 `border` 参数。例如，下面的调用将添加窗口部件到 `sizer` 列表的尾部，并使该窗口部件的周围有5个像素宽度的边框：

```
sizer.Add(widget, 0, wx.ALL | wx.EXPAND, 5)
```

该部件然后将扩展以填充它的有效空间，且四周始终留有5个像素的空白。

使用其它类型的sizer

我们已经讨论了基本的 `sizer` ，现在我们可以转向更复杂和更灵活的 `sizer` 了。其中两个（ `flex` `grid` `sizer` 和 `grid` `bag` `sizer` ）本质上是 `grid` 的变种。另外两个（ `box` 和 `static` `box` `sizer` ）使用一个不同的和更灵活的布局结构。

什么是flex grid sizer？

`flex grid sizer`是`grid sizer`的一个更灵活的版本。它与标准的`grid sizer`几乎相同，除了下面的例外：

1、每行和每列可以有各自的尺寸。

2、默认情况下，当尺寸调整时，它不改变它的单元格的尺寸。如果需要的话，你可以指定哪行或哪列应该增长。

3、它可以在两个方向之一灵活地增长，意思是你可以为个别的子元素指定比列量，并且你可以指定固定方向上的行为。

图11.7显示了一个`flex grid sizer`，它的布局也是9个单元格。这里的中间单元格更大。

图**11.7**

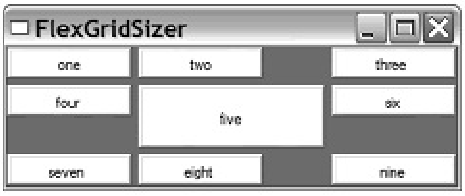


Figure 11.7 A simple flex grid sizer

与图11.5相比较，对于相同的布局，图11.5中每个单元格的尺寸与中间对象的相同，在`flex grid sizer`中，单元格的尺寸大小根据它所在的行和列来定。它们宽度是该列中宽度最大的项目的宽度，它们的高度是该行中宽度最高的项目的宽度。在这里，项目“`four`”和项目“`six`”的单元格的高度比项目本身的高度更高，因为其同行中的项目“`five`”，而"`two`"和"`seven`"的单元格的宽度也更宽。“`one`,”“`three`,”“`seven`,”和“`nine`”的单元格是正常的尺寸，并且不受较大的窗口部件的影响。

图11.8展示了当调整窗口尺寸时，`flex grid sizer`的默认行为——单元格的尺寸不改变。

图**11.8**

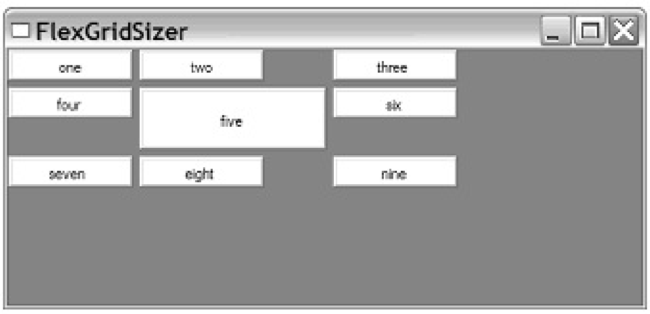


Figure 11.8
A flex grid sizer being resized.

例11.6显示了产生了图11.8的代码

例**11.6** 创建一个**`flex grid sizer`**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "FlexGridSizer")
        sizer = wx.FlexGridSizer(rows=3, cols=3, hgap=5, vgap=5)
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, 0)
        center = self.FindWindowByName("five")
        center.SetMinSize((150,50))
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

一个 `flex` `grid` `sizer` 是 `wx.FlexGridSizer` 的一个实例。类 `wx.FlexGridSizer` 是 `wx.GridSizer` 的子类，所以 `wx.GridSizer` 的属性方法依然有效。 `wx.FlexGridSizer` 的构造函数与其父类的相同：

`wx.FlexGridSizer(rows` , `cols` , `vgap` , `hgap)`

为了当 `sizer` 扩展时，使一行或列也扩展，你需要使用适当的方法显式地告诉该 `sizer` 该行或列是可扩展的：

```
AddGrowableCol(idx, proportion=0)
AddGrowableRow(idx, proportion=0)
```

当 `sizer` 水平扩展时，关于新宽度的默认行为被等同地分配给每个可扩展的列。同样，一个垂直的尺寸调整也被等同地分配给每个可扩展的行。要改变这个默认的行为并且使不同的行和列有不现的扩展比率，你需要使用 `proportion` 参数。如果 `proportion` 参数被使用了，那么与该参数相关的新的空间就被分配给了相应的行或列。例如，如果你有两个尺寸可调整的行，并且它们的 `proportion` 分别是2和1，那么这第一个行将得到新空间的2/3，第二行将得到1/3。图11.9显示使用 `proportional` （比列）空间的 `flex` `grid` `sizer` 。在这里，中间行和列所占的比例是2和5，两端的行和列所占的比例是1。

图**11.9**

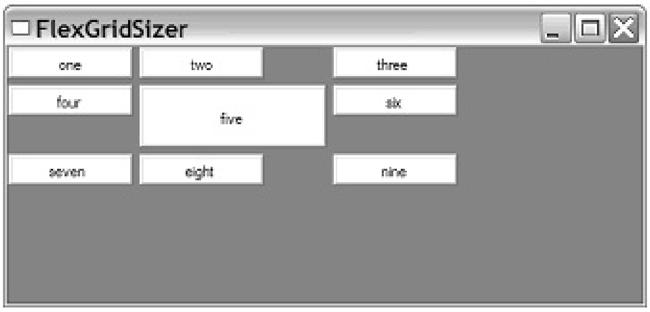


Figure 11.8
A flex grid sizer being resized.

正如你可以看到的，当所有的单元格增大时，中间的行和列的增大是两端的两倍。窗口部件的没有改变尺寸以填表充它们的单元格，虽然可以通过在当它们被添加到 `sizer` 时使用 `wx.EXPAND` 来实现。例11.7显示了产生图11.9的代码。

例**11.7**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Resizing Flex Grid Si
zer")
        sizer = wx.FlexGridSizer(rows=3, cols=3, hgap=5, vgap=5)
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, 0)
        center = self.FindWindowByName("five")
        center.SetMinSize((150,50))
        sizer.AddGrowableCol(0, 1)
        sizer.AddGrowableCol(1, 2)
        sizer.AddGrowableCol(2, 1)
        sizer.AddGrowableRow(0, 1)
        sizer.AddGrowableRow(1, 5)
        sizer.AddGrowableRow(2, 1)
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

如果你对一个可扩展的行或列使用了比例尺寸，那么你需要对该方向上的所有可扩展的行或列指定一个比例量，否则你将得到一个糟糕的效果。

在 `flex grid sizer` 中还有另外一个机制用于控制窗口部件的增长（执不执行先前 `AddGrowable` *方法的设置）。默认情况下，比例尺寸适用于 `flex grid` 的两个方向；但是你可以通过使用 `SetFlexibleDirection(direction)` 方法来指定仅某个方向应该按比例调整尺寸，参数 `direction` 的值可以是： `wx.HORIZONTAL` ， `wx.VERTICAL` ，或 `wx.BOTH` （默认值）。然后你可以使用 `SetNonFlexibleGrowMode(mode)` 方法来指定另一个方向上的行为。例如，如果你调用

了 `SetFlexibleDirection(wx.HORIZONTAL)` 方法，列的行为就遵循 `AddGrowableCol()` ，然后调用 `SetNonFlexibleGrowMode()` 来定义行的行为。表11.3显示了 `mode` 参数的有效值。

表**11.3**

`wx.FLEX_GROWMODE_ALL` ： `flex` `grid` 在没有使用 `SetFlexibleDirection` 的方向上等同地调整所有单元格的尺寸。这将覆盖使用 *`AddGrowable`* 方法设置的任何行为——所有的单元格都将被调整尺寸，不管它们的比例或它们是否被指定为可扩展（增长）的。

`wx.FLEX_GROWMODE_NONE` ：在没有使用 `SetFlexibleDirection` *的方向上的单元格的尺寸不变化，不管它们是否被指定为可增长的。

`wx.FLEX_GROWMODE_SPECIFIED` ：在没有使用 `SetFlexibleDirection` *的方向上，只有那些可增长的单元格才增长。但是 `sizer` 将忽略任何的比例信息并等同地增长那些单元格。这是一个默认行为。

上面段落中所讨论的 `SetFlexibleDirection` 和 `SetNonFlexibleGrowMode` 方法都有对应的方法： `GetFlexibleDirection()` 和 `GetNonFlexibleGrowMode()` ，它们返回整型标记。在上表中要强调的是，任何使用这些方法来指定的设置将取代通过 `AddGrowableCol()` 和 `AddGrowableRow()` 创建的设置。

什么是**grid bag sizer?**

`grid` `bag` `sizer` 是对 `flex` `grid` `sizer` 进一步的增强。在 `grid` `bag` `sizer` 中有两个新的变化：

1、能够将一个窗口部件添加到一个特定的单元格。 2、能够使一个窗口部件跨越几个单元格（就像 `HTML` 表单中的表格所能做的一样）。

图**11.10**

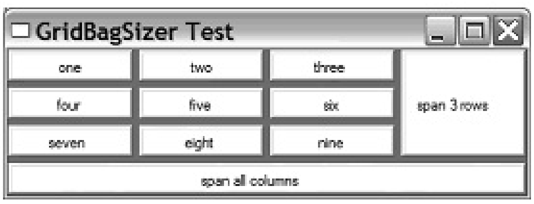


Figure 11.10
A sample grid bag sizer

图11.10显示了一个 `grid` `bag` `sizer` 的示例。它与本章前面的例子很相似，只是增加了新的窗口部件以展示跨行和跨列。

例11.8显示了产生图11.10的代码。注意这里的 `Add()` 方法与以前的看起来有点不同。

例**11.8** **`Grid` `bag` `sizer`** 示例代码

```
#coding=utf-8
#!python
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "GridBagSizer Test")
        sizer = wx.GridBagSizer(hgap=5, vgap=5)
        for col in range(3):
            for row in range(3):
                bw = BlockWindow(self, label=labels[row*3 + col]
)
                sizer.Add(bw, pos=(row,col))
        # 跨行
        bw = BlockWindow(self, label="span 3 rows")
        sizer.Add(bw, pos=(0,3), span=(3,1), flag=wx.EXPAND)
        # 跨列
        bw = BlockWindow(self, label="span all columns")
        sizer.Add(bw, pos=(3,0), span=(1,4), flag=wx.EXPAND)
        # 使最后的行和列可增长
        sizer.AddGrowableCol(3)
        sizer.AddGrowableRow(3)
        self.SetSizer(sizer)
        self.Fit()
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

`grid` `bag` `sizer` 是 `wx.GridBagSizer` 的实例， `wx.GridBagSizer` 是 `wx.FlexGridSizer` 的一个子类。这意味着所有 `flex` `grid` `sizer` 的属性， `grid` `bag` `sizer` 都适用。

`wx.GridBagSizer` 的构造函数与它的父类有点不同：

```
wx.GridBagSizer(vgap=0, hgap=0)
```

在一个 `grid` `bag` `sizer` 中，你不必去指定行和列的数量，因为你可以直接将子项目添加进特定的单元格—— `sizer` 将据此计算出网格的尺度。

在 `grid` `bag` `sizer` 上使用 `Add()` 方法

对于 `grid` `bag` `sizer` ， `Add()` 方法与别的 `sizer` 不同。它有四个可选的形式：

```
1 Add(window, pos, span=wx.DefaultSpan, flag=0, border=0,
userData=None)
2 Add(sizer, pos, span=wx.DefaultSpan, flag=0, border=0,
userData=None)
3 Add(size, pos, span=wx.DefaultSpan, flag=0, border=0,
userData=None)
4 AddItem(item)
```

这些看起来应该很熟悉，在运行上也与通常的 `sizer` 的方法相似。 `window` , `sizer` , `size` , `flag` , `border` ,和 `userData` 参数的行为与通常 `sizer` 的方法中的是相同的。 `pos` 参数代表 `sizer` 中的窗口部件要赋予的单元格。技术上讲， `pos` 参数是类 `wx.GBPosition` 的一个实例，但是通过 `wxPython` 变换，你可以仅传递一个(行,列)形式的元组， `grid` `bag` 的左上角是(0,0)。

同样， `span` 参数代表窗口部件应该占据的行和列的数量。它是类 `wx.GBSpan` 的一个实例，但是， `wxPython` 也使你能够传递一个(行的范围，列的范围)形式的元组。如果跨度没有指定，那么默认值是(1,1)，这意味该窗口部件在两个方向都只能占据一个单元格。例如，要在第二行第一列放置一个窗口部件，并且使它占据三行两列，那么你将这样调用： `Add(widget` , (1, 0), (3, 2))（索引是从0开始的）。

`Additem` 方法的 `item` 参数是类 `wx.GBSizerItem` 的一个实例，它包含了 `grid` `bag` `sizer` 放置项目所需要的全部信息。你不太可能需要直接去创建一个 `wx.GBSizerItem` 。如果你想去创建一个，那么它的构造函数的参数与 `grid` `bag` `sizer` 的其它 `Add()` 方法相同。一旦你有了一个 `wx.GBSizerItem` ，这儿有许多的 `get` *方法使你能够访问项目的属性，也许这最有用的是 `GetWindow()` ，它返回实际显示的窗口部件。

由于项目是使用行和列的索引以及跨度被添加到一个 `grid` `bag` `sizer` 的，所以项目被添加的顺序不必按照其它 `sizer` 所要求的对应它们的显示顺序。这使得跟踪哪个项实际显示在哪个单元格有一点头痛。表11.4列出了几个方法， `grid` `bag` `sizer` 通过它们来使你对项目的跟踪较为容易。

表11.4 `Grid` `bag` `sizer` 管理项目的方法

`CheckForIntersection(item` , `excludeItem = None)`
`CheckForIntersection(pos` , `span` , `excludeItem = None)` ：将给定的项目或给定的位置和跨度同 `sizer` 中的项目进行比对。如果有任一项与给定项目的位置或给定的位置和跨度重叠，则返回 `True` 。 `excludeItem` 是一个可选的项，它不被包括在比对中（或许是因为它正在测试中）。 `pos` 参数是一个 `wx.GBPosition` 或一个元组。 `span` 参数是一个 `wx.GPSpan` 或一个元组。

`FindItem(window)` `FindItem(sizer)` ：返回对应于给定的窗口或 `sizer` 的 `wx.GBSizerItem` 。如果窗口或 `sizer` 不在 `grid` `bag` 中则返回 `None` 。这个方法不递归检查其中的子 `sizer` 。

`FindItemAtPoint(pt)` ： `pt` 参数是对应于所包含的框架的坐标的一个 `wx.Point` 实例或一个 `Python` 元组。这个方法返回位于该点的 `wx.GBSizerItem` 。如果这个位置在框架的边界之外或如果该点没

有 `sizer` 项目，则返回 `None` 。

`FindItemAtPosition(pos)` ：该方法返回位于给定单元格位置的 `wx.GBSizerItem` ，参数 `pos` 是一个 `wx.GBPosition` 或一个 `Python` 元组。如果该位置在 `sizer` 的范围外或该位置没有项目，则返回 `None` 。

`FindItemWithData(userData)` ：返回 `grid` `bag` 中带有给定的 `userData` 对象的一个项目的 `wx.GBSizerItem` 。

`Grid` `bag` 也有一对能够用于处理单元格尺寸和项目位置的属性。在 `grid` `bag` 被布局好并显示在屏幕上后，你可以使用方法 `GetCellSize(row , col)` 来获取给定的单元格显示在屏幕上的尺寸。这个尺寸包括了由 `sizer` 本身所管理的水平和垂直的间隔。你可以使用方法 `GetEmptyCellSize()` 得到一个空单元格的尺寸，并且你可以使用 `SetEmptyCellSize(sz)` 改变该属性，这里的 `sz` 是一个 `wx.Size` 对象或一个 `Python` 元组。

你也可以使用方法 `GetItemPosition()` 和 `GetItemSpan()` 来得到 `grid` `bag` 中的一个对象的位置或跨度。这两个方法要求一个窗口，一个 `sizer` 或一个索引作为参数。这个索引参数与 `sizer` 的 `Add()` 列表中的索引相对应，这个索引在 `grid` `bag` 的上下文中没多大意思。上面的两个 `get` 方法都有对应的 *set* 方法， `SetItemPosition(window , pos)` 和 `SetItemSpan(window , span)` ，这两个方法的第一个参数可以是 `window` , `sizer` ,或 `index` ，第二个参数是一个 `Python` 元组或一个 `wx.GBPosition` 或 `wx.GBSpan` 对象。

什么是box

`sizer` ？

`box` `sizer` 是 `wxPython` 所提供的 `sizer` 中的最简单和最灵活的 `sizer` 。一个 `box` `sizer` 是一个垂直列或水平行，窗口部件在其中从左至右或从上到下布置在一条线上。虽然这听起来好像用处太简单，但是来自相互之间嵌套 `sizer` 的能力使你能够在每行或每列很容易放置不同数量的项目。由于每个 `sizer` 都是一个独立的实体，因此你的布局就有了更多的灵活性。对于大多数的应用程序，一个嵌套有水平 `sizer` 的垂直 `sizer` 将使你能够创建你所需要的布局。

图11.11-11.14展示了几个简单的 `box` `sizer` 的例子。图中所展示的各个框架我们都是手动调整了大小的，以便展示每个 `sizer` 是如何响应框架的增大的。图11.11显示了一个水平的 `box` `sizer` ，图11.12在一个垂直的 `box` `sizer` 显示了现图11.11相同的窗口部件。

图11.11



Figure 11.11 A horizontal box sizer

图11.12

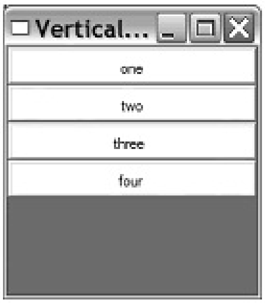


Figure 11.12
A vertical box sizer

图11.13显示了一个垂直的`sizer`，其中一个窗口部件被设置成可扩展并填充有效的垂直空间。图11.14展示了一个垂直的`sizer`，其中的两个窗口部件设置为按不同的比例占据有效的垂直空间。

图**11.13**



Figure 11.13
A vertical sizer
with one stretch
element

图**11.14**



Figure 11.14
A vertical sizer
with two stretch
elements

生成以上四个`sizer`框架的示例代码如例11.9所示。

例**11.9** 产生多个**`box`** **`sizer`**

```
import wx
from blockwindow import BlockWindow
labels = "one two three four".split()
class TestFrame(wx.Frame):
    title = "none"
```

```
    def __init__(self):
        wx.Frame.__init__(self, None, -1, self.title)
        sizer = self.CreateSizerAndWindows()
        self.SetSizer(sizer)
        self.Fit()
class VBoxSizerFrame(TestFrame):
    title = "Vertical BoxSizer"
    def CreateSizerAndWindows(self):
        sizer = wx.BoxSizer(wx.VERTICAL)
        for label in labels:
            bw = BlockWindow(self, label=label, size=(200,30))
            sizer.Add(bw, flag=wx.EXPAND)
        return sizer
class HBoxSizerFrame(TestFrame):
    title = "Horizontal BoxSizer"
    def CreateSizerAndWindows(self):
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        for label in labels:
            bw = BlockWindow(self, label=label, size=(75,30))
            sizer.Add(bw, flag=wx.EXPAND)
        return sizer
class VBoxSizerStretchableFrame(TestFrame):
    title = "Stretchable BoxSizer"
    def CreateSizerAndWindows(self):
        sizer = wx.BoxSizer(wx.VERTICAL)
        for label in labels:
            bw = BlockWindow(self, label=label, size=(200,30))
            sizer.Add(bw, flag=wx.EXPAND)
        # Add an item that takes all the free space
        bw = BlockWindow(self, label="gets all free space", size
=(200,30))
        sizer.Add(bw, 1, flag=wx.EXPAND)
        return sizer
class VBoxSizerMultiProportionalFrame(TestFrame):
    title = "Proportional BoxSizer"
    def CreateSizerAndWindows(self):
        sizer = wx.BoxSizer(wx.VERTICAL)
        for label in labels:
            bw = BlockWindow(self, label=label, size=(200,30))
            sizer.Add(bw, flag=wx.EXPAND)
        # Add an item that takes one share of the free space
        bw = BlockWindow(self,
                label="gets 1/3 of the free space",
                size=(200,30))
        sizer.Add(bw, 1, flag=wx.EXPAND)
        # Add an item that takes 2 shares of the free space
        bw = BlockWindow(self,
                label="gets 2/3 of the free space",
                size=(200,30))
        sizer.Add(bw, 2, flag=wx.EXPAND)
        return sizer
app = wx.PySimpleApp()
frameList = [VBoxSizerFrame, HBoxSizerFrame,
```

```
                VBoxSizerStretchableFrame,
                VBoxSizerMultiProportionalFrame]
for klass in frameList:
    frame = klass()
    frame.Show()
app.MainLoop()
```

`box sizer` 是类 `wx.BoxSizer` 的实例， `wx.BoxSizer` 是 `wx.Sizer` 的子类，相对于 `wx.Sizer` ， `wx.BoxSizer` 几乎没有增加新的方法。 `wx.BoxSizer` 的构造函数有一个参数：

```
wx.BoxSizer(orient)
```

参数 `orient` 代表该 `sizer` 的方向，它的取值可以是 `wx.VERTICAL` 或 `wx.HORIZONTAL` 。对于 `box sizer` 所定义的唯一一个新的方法是 `GetOrientation()` ，它返回构造函数中 `orient` 的整数值。一旦一个 `box sizer` 被创建后，你就不能改变它的方向了。 `box sizer` 的其它的函数使用本章早先所讨论的一般的 `sizer` 的方法。

`box sizer` 的布局算法对待该 `sizer` 的主方向（当构建的时候已被它的方向参数所定义）和次要方向是不同的。特别地， `proportion` 参数只适用于当 `sizer` 沿主方向伸缩时，而 `wx.EXPAND` 标记仅适用于当 `sizer` 的尺寸在次方向上变化时。换句话说，当一个垂直的 `box sizer` 被垂直地绘制时，传递给每个 `Add()` 方法调用的参数 `proportion` 决定了每个项目将如何垂直地伸缩。除了影响 `sizer` 和它的项目的水平增长外，参数 `proportion` 以同样的方式影响水平的 `box sizer` 。在另一方面，次方向的增长是由对项目所使用的 `wx.EXPAND` 标记来控制的，所以，如果它们设置了 `wx.EXPAND` 标记的话，在一个垂直的 `box sizer` 中的项目将只在水平方向增长。否则这些项目保持它们的最小或最合适的尺寸。图6.7演示了这个过程。

在 `box sizer` 中，项目的比例增长类似于 `flex grid sizer` ，但有一些例外。第一， `box sizer` 的比例行为是在窗口部件被添加到该 `sizer` 时，使用 `proportion` 参数被确定的——你无需像 `flex grid sizer` 那样单独地指定它的增长性。第二，比例为0的行为是不同的。在 `box sizer` 中，0比例意味着该窗口部件在主方向上不将根据它的最小或最合适尺寸被调整尺寸，但是如果 `wx.EXPAND` 标记被使用了的话，它仍可以在次方向上增长。当 `box sizer` 在主方向上计算它的项目的布局时，它首先合计固定尺寸的项目所需要的空间，这些固定尺寸的项目，它们的比例为0。余下的空间按项目的比例分配。

什么是static

`box sizer` ?

一个 `static box sizer` 合并了 `box sizer` 和静态框（ `static box` ），静态框在 `sizer` 的周围提供了一个漂亮的边框和文本标签。图11.15显示了三个 `static box sizer` 。

图**11.15**

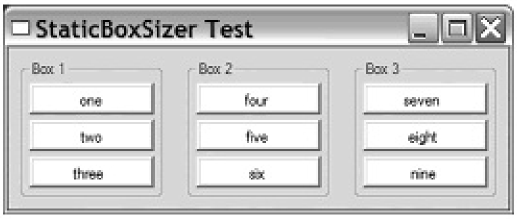


Figure 11.15
Three static box sizers

例11.10显示了产生上图的代码。这里有三个值得注意的事件。首先你必须单独于`sizer`创建静态框对象，第二是这个例子展示了如何使用嵌套的`box sizer`。本例中，三个垂直的`static box sizer`被放置于一个水平的`box sizer`中。

例**11.10** **`static box sizer`**的一个例子

```
import wx
from blockwindow import BlockWindow
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "StaticBoxSizer Test")
        self.panel = wx.Panel(self)
        # make three static boxes with windows positioned inside
 them
        box1 = self.MakeStaticBoxSizer("Box 1", labels[0:3])
        box2 = self.MakeStaticBoxSizer("Box 2", labels[3:6])
        box3 = self.MakeStaticBoxSizer("Box 3", labels[6:9])
        # We can also use a sizer to manage the placement of oth
er
        # sizers (and therefore the windows and sub-sizers that
they
        # manage as well.)
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(box1, 0, wx.ALL, 10)
        sizer.Add(box2, 0, wx.ALL, 10)
        sizer.Add(box3, 0, wx.ALL, 10)
        self.panel.SetSizer(sizer)
        sizer.Fit(self)
    def MakeStaticBoxSizer(self, boxlabel, itemlabels):
        # first the static box
        box = wx.StaticBox(self.panel, -1, boxlabel)
        # then the sizer
        sizer = wx.StaticBoxSizer(box, wx.VERTICAL)
        # then add items to it like normal
        for label in itemlabels:
            bw = BlockWindow(self.panel, label=label)
            sizer.Add(bw, 0, wx.ALL, 2)
        return sizer
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

`static` `box` `sizer` 是类 `wx.StaticBoxSizer` 的实例， `wx.StaticBoxSizer` 是 `wx.BoxSizer` 的子类。它的构造函数要求的参数是静态框和方向：

`wx.StaticBoxSizer(box` , `oient)`

在这个构造函数中， `orient` 的意义与原 `wx.BoxSizer` 相同， `box` 参数是一个 `wx.StaticBox` 。对于 `static` `box` `sizer` 所定义的别的方法只有一个： `GetStaticBox()` ，它返回用于建造该 `sizer` 的 `wx.StaticBox` 。一旦该 `sizer` 被创建，那么你就不能再改变这个静态框了。

`wx.StaticBox` 类有一个用于 `wxPython` 控件的标准的构造函数，但是其中许多参数都有默认值，可以忽略。

`wx.StaticBox(parent` , `id` , `label` , `pos` = `wx.DefaultPosition` ,

- `size` = `wx.DefaultSize` , `style` =0, `name` =" `staticBox` ")

使用一个 `static` `box` `sizer` ，你不需要去设置 `pos` , `size` , `style` , 或 `name` 参数，因为位置和尺寸将由 `sizer` 管理，并且没用单独用于 `wx.StaticBox` 的样式。这使得构造更简单：

`box` = `wx.StaticBox(self.panel` , -1, `boxlabel)`

到目前为止，我们已经展示了各种 `sizer` ，我们将给你展示如何在实际的布局中使用它们。对于另外一个用于创建一个复杂布局的例子，请参看第六章。

一个现实中使用sizer的例子

迄今为止，我们所展示的有关 `sizer` 的例子都是在显示它们的功能方面。下面，我们将展示一个如何使用 `sizer` 来建造一个真实的布局。图11.16显示了一个使用不同 `sizer` 建造的复杂程度适中的布局。

图**11.16**



Figure 11.16
A more realistic sizer example

例11.11显示了产生上图的代码。这段代码看起来有点复杂，但是我们将对它分块解读。

例**11.11** 用 **`sizer`** 来建造地址表单

```
#coding=utf-8
#!python
import wx
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Real World Test")
        panel = wx.Panel(self)
        # First create the controls
        topLbl = wx.StaticText(panel, -1, "Account Information")
#1 创建窗口部件
        topLbl.SetFont(wx.Font(18, wx.SWISS, wx.NORMAL, wx.BOLD)
)
        nameLbl = wx.StaticText(panel, -1, "Name:")
```

```
        name = wx.TextCtrl(panel, -1, "");
        addrLbl = wx.StaticText(panel, -1, "Address:")
        addr1 = wx.TextCtrl(panel, -1, "");
        addr2 = wx.TextCtrl(panel, -1, "");
        cstLbl = wx.StaticText(panel, -1, "City, State, Zip:")
        city  = wx.TextCtrl(panel, -1, "", size=(150,-1));
        state = wx.TextCtrl(panel, -1, "", size=(50,-1));
        zip   = wx.TextCtrl(panel, -1, "", size=(70,-1));
        phoneLbl = wx.StaticText(panel, -1, "Phone:")
        phone = wx.TextCtrl(panel, -1, "");
        emailLbl = wx.StaticText(panel, -1, "Email:")
        email = wx.TextCtrl(panel, -1, "");
        saveBtn = wx.Button(panel, -1, "Save")
        cancelBtn = wx.Button(panel, -1, "Cancel")
        # Now do the layout.
        # mainSizer is the top-level one that manages everything
#2 垂直的sizer
        mainSizer = wx.BoxSizer(wx.VERTICAL)
        mainSizer.Add(topLbl, 0, wx.ALL, 5)
        mainSizer.Add(wx.StaticLine(panel), 0,
                wx.EXPAND|wx.TOP|wx.BOTTOM, 5)
        # addrSizer is a grid that holds all of the address info
#3 地址列
        addrSizer = wx.FlexGridSizer(cols=2, hgap=5, vgap=5)
        addrSizer.AddGrowableCol(1)
        addrSizer.Add(nameLbl, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        addrSizer.Add(name, 0, wx.EXPAND)
        addrSizer.Add(addrLbl, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        addrSizer.Add(addr1, 0, wx.EXPAND)
#4 带有空白空间的行
        addrSizer.Add((10,10)) # some empty space
        addrSizer.Add(addr2, 0, wx.EXPAND)
        addrSizer.Add(cstLbl, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        # the city, state, zip fields are in a sub-sizer
#5 水平嵌套
        cstSizer = wx.BoxSizer(wx.HORIZONTAL)
        cstSizer.Add(city, 1)
        cstSizer.Add(state, 0, wx.LEFT|wx.RIGHT, 5)
        cstSizer.Add(zip)
        addrSizer.Add(cstSizer, 0, wx.EXPAND)
#6 电话和电子邮箱
        addrSizer.Add(phoneLbl, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        addrSizer.Add(phone, 0, wx.EXPAND)
        addrSizer.Add(emailLbl, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        addrSizer.Add(email, 0, wx.EXPAND)
        # now add the addrSizer to the mainSizer
#7 添加Flex sizer
        mainSizer.Add(addrSizer, 0, wx.EXPAND|wx.ALL, 10)
```

```
        # The buttons sizer will put them in a row with resizeab
le
        # gaps between and on either side of the buttons
#8 按钮行
        btnSizer = wx.BoxSizer(wx.HORIZONTAL)
        btnSizer.Add((20,20), 1)
        btnSizer.Add(saveBtn)
        btnSizer.Add((20,20), 1)
        btnSizer.Add(cancelBtn)
        btnSizer.Add((20,20), 1)
        mainSizer.Add(btnSizer, 0, wx.EXPAND|wx.BOTTOM, 10)
        panel.SetSizer(mainSizer)
        # Fit the frame to the needs of the sizer.  The frame wi
ll
        # automatically resize the panel as needed.  Also preven
t the
        # frame from getting smaller than this size.
        mainSizer.Fit(self)
        mainSizer.SetSizeHints(self)
app = wx.PySimpleApp()
TestFrame().Show()
app.MainLoop()
```

#1 代码的第一部分是创建使用在窗口中的窗口部件，它们在这行开始。我们在增加 `sizer` 前将它们全部创建。

#2 在这个布局中的主 `sizer` 是 `mainSizer` ，它是一个竖直的 `box sizer` 。被添加到 `mainSizer` 的第一个元素是顶部的静态文本标签和一个 `static line` 。

#3 在 `box sizer` 中接下来的元素是 `addrSizer` ，它是一个 `flex grid sizer` ，它有两列，它两列用于容纳其余的地址信息。 `addrSizer` 的左列被设计用于静态文本标签，而右列用于得到文本控件。这意味着标签和控件需要被交替的添加，以保证 `grid` 的正确。你可以看到 `nameLbl` ， `name` ， `addrLbl` ，和 `addr1` 是首先被添加到该 `flex grid` 中的四个元素。

#4 这接下来的行是不同的，因为这第二个地址行没有标签，一个(10,10)尺寸的空白块被添加，然后是 `addr2` 控件。

#5 接下来的行又有所不同，包括“ `City` ， `State` ， `Zip` ”的行要求三个不同的文本控件，基于这种情况，我们创建了一个水平的 `box sizer` ： `cstSizer` 。这三个控件被添加给 `cstSizer` ，然后这个 `box sizer` 被添加到 `addrSizer` 。

#6 电话和电子邮箱行被添加到 `flex sizer` 。

#7 有关地址的 `flex sizer` 被正式添加到主 `sizer` 。

#8 按钮行作为水平 `box sizer` 被添加，其中有一些空白元素以分隔按钮。

在调整了 `sizer` （ `mainSizer.Fit(self)` ）和防止框架变得更小之后（ `mainSizer.SetSizeHints(self)` ），元素的布局就结束了。

在读这接下来的段落或运行这个例子之前，请试着想出该框架将会如何在水平和竖直方向上响应增长。

如果该窗口在竖直方向上的大小改变了，其中的元素不会移动。这是因为主 sizer 是一个垂直的 box sizer ，你是在它的主方向上改变尺寸，它没有一个顶级元素是以大于0的比列被添加的。如果这个窗口在水平方向被调整尺寸，由于这个主 sizer 是一个垂直的 box sizer ，你是在它的次方向改变尺寸，因此它的所有有 wx.EXPAND 标记的元素将水平的伸展。这意味着顶部的标签不增长，但是 static line 和子 sizer 将水平的增长。用于地址的 flex grid sizer 指定列1是可增长的，这意味着包含文本控件的第二列将增长。在“ City , State , Zip ”行内，比列为1的 city 元素将伸展，而 state 和 ZIP 控件将保持尺寸不变。按钮将保持原有的尺寸，因为它们的比列是0，但是按钮所在行的空白空间将等分地占据剩下的空间，因为它们每个的比列都是1。

因此如果你想出的窗口伸展的样子和下图11.17相同，那么你是正确的。

图**11.17**

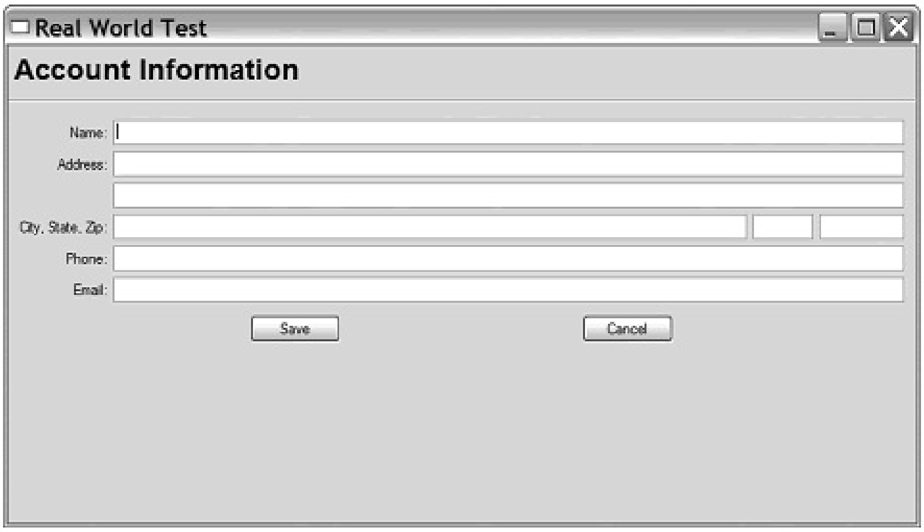


Figure 11.17 The account information, after the window was stretched

本章小结

1、 Sizer 是对 wxPython 程序中管理布局问题的解决方法。不用手动指定每个元素在布局中的位置和尺寸，你可以将元素添加到一个 sizer 中，由 sizer 负责将每个元素放置到屏幕上。当用户调整框架的尺寸时， sizer 管理布局是相当好的。

2、所有的 wxPython 的 sizer 都是类 wx.Sizer 的一个子类的实例。要使用一个 sizer ，你需要把它与一个容器型的窗口部件关联起来。然后，对于要添加到容器中的子窗口部件，你也必需将它们添加到该 sizer 。最后，调用该 sizer 的 Fit() 方法来触发该 sizer 的算法，以便布局和放置。

3、所有的`sizer`开始都给它们的孩子以最小的尺寸。每种`sizer`各自使用不同的机制来放置窗口部件，所以相同组的窗口部件放置在不同的`sizer`中时，它们的外观也是不同的。

4、或许在`wxPython`中，最简单的`sizer`是`grid sizer(wx.GridSizer)`。在`grid sizer`中，元素按照它们被添加给`sizer`的顺序被放置在一个二维的网格中，按照从左到右，从上到下的方式排列。通常你负责设定列数，`sizer`确定行数。你能够同时指定行和列，如果你愿意的话。

5、所有的`sizer`都有各自不同的用来将窗口部件添加到`sizer`的方法。由于窗口部件添加到`sizer`中的顺序对于最后的布局是重要的，所以有不同的方法用来添加一个新的窗口部件到列表中的前面、后面或任意点。在一个窗口部件被添加到`sizer`时，另外的属性可以被设置，它们控制当`sizer`增减时，其中的子元素如何变化。`sizer`也能够被配置来在对象的某些或全部边上放置一个边界间隙。

第十二章 操作基本图像

本章内容：

- 装载图像和创建图像对象
- 创建设备上下文
- 绘制到设备上下文
- 绘制文本到设备上下文
- 管理画刷、画笔和设备坐标

任何用户界面工具的最基本的行为就是在屏幕上绘制。在最基本的层面上来说，定义在 `wxPython` 中的每个窗口部件都是由发送给屏幕的一系列命令构成的。那些绘制命令是否是在 `wxPython` 代码库中，这依赖于相关窗口部件对于本地操作系统是否是本地化的或完全由 `wxPython` 所定义的。在这一章，我们将给你展示如何在基本绘制命令层面上控制 `wxPython` 。我们也将给你展示如何管理和显示其它的图形化元素如图像和字体。

被 `wxPython` 用于绘制的主要概念是设备上下文（ `device context` ）。设备上下文使用一个标准的 `API` 来管理对设备（如屏幕或打印机）的绘制。设备上下文类用于最基本的绘制功能，如绘制一条直线、曲线或文本。

使用图像工作

大多数的应用程序都需要载入至少一个图像，这些图像存储在外部的文件中。样例包括工具栏图片、光标、图标、启始画面或仅仅用于装饰以增加一些时髦感的图像。传统上，使用图像工作的复杂性是不得不处理用于储存图像的不同的图片文件格式。幸运的是， `wxPython` 内部为你做了所有的这些。你将使用相同的抽象概念来处理任何图像，而不用关心它的原始格式。

在随后的几节中，我们将讨论 `wxPython` 用来管理图像的那么抽象概念，这包括大尺寸图像(`large - scale images)` ，以及光标图像。你将看到如何装载图像到你的程序中，然后如何操作它们。

如何载入图像？

在 `wxPython` 中，图像处理是一个双主管系统，与平台无关的图像处理由类 `wx.Image` 管理，而与平台有关的图像处理由类 `wx.Bitmap` 管理。实际上，意思就是外部文件格式由 `wx.Image` 装载和保存，而 `wx.Bitmap` 负责将图像显示到屏幕。图12.1显示了不同图像和位图的创建，按照不同的文件类型读入。

要从一个文件载入一个图像，使用 `wx.Image` 的构造函数：

```
wx.Image(name, type=wx.BITMAP_TYPE_ANY, index=-1)
```

图**12.1**

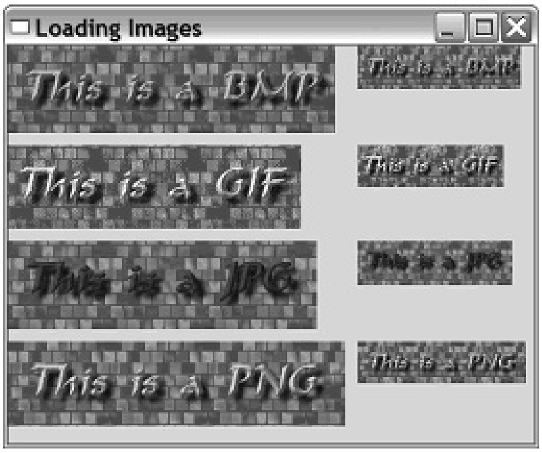


Figure 12.1
Big and little images
of different types

参数 `name` 是图像文件的名字，参数 `type` （类型）是处理器类型。 `type` 的 `ID` 可以是 `wx.BITMAP_TYPE_ANY` 或表12.1中的一个。如果你使用 `wx.BITMAP_TYPE_ANY` ，那么 `wxPython` 将试图自动检测该文件的类型。如果你使用一个特定的文件类型，那么 `wxPython` 将使用该类型转换这个文件。例12.1显示了如何使用 `wx.BITMAP_TYPE_ANY` 来载入图像。

例**12.1** 载入并缩放简单图像

```
import wx

filenames = ["image.bmp", "image.gif", "image.jpg", "image.png"
]

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Loading Images")
        p = wx.Panel(self)

        fgs = wx.FlexGridSizer(cols=2, hgap=10, vgap=10)
        for name in filenames:
            #1 从文件载入图像
            img1 = wx.Image(name, wx.BITMAP_TYPE_ANY)

            # Scale the oiginal to another wx.Image
            w = img1.GetWidth()
            h = img1.GetHeight()
            img2 = img1.Scale(w/2, h/2)#2 缩小图像

            #3 转换它们为静态位图部件
            sb1 = wx.StaticBitmap(p, -1, wx.BitmapFromImage(img1
))
            sb2 = wx.StaticBitmap(p, -1, wx.BitmapFromImage(img2
))

            # and put them into the sizer
            fgs.Add(sb1)
            fgs.Add(sb2)

        p.SetSizerAndFit(fgs)
        self.Fit()

app = wx.PySimpleApp()
frm = TestFrame()
frm.Show()
app.MainLoop()
```

上面的代码应该很简单。代码开始是我们想要载入的图像文件的名字，我们使用 `wx.BITMAP_TYPE_ANY` 类型标记指示 `wxPython` 去断定图像文件的格式，而用不着我们操心。然后我们使用图像的方法将图像缩小一半，并将图像转换为位图。

你也可以显示地指定图像文件的格式，在下一节，我们将显示 `wxPython` 所支持的图像文件格式。

指定一个图像文件格式

图像由所用的图像处理器管理。一个图像处理器是 `wx.ImageHandler` 的一个实例，它为管理图像格式提供了一个插入式的结构。在通常的情况下，你不需要考虑图像处理器是如何工作的。你所需要知道的是每个处理器都有它自己唯一

的 wxPython 标识符，用以载入相关格式的文件。表12.1列出了所支持的格式。

表12.1 wxPython 支持的图像文件格式

处理器类	类型标记	说明
wx.ANIHandler	wx.BITMAP_TYPE_ANI	动画光标格式。这个处理器只载入图像而不保存它们。
wx.BMPHandler	wx.BITMAP_TYPE_BMP	Windows 和 OS /2位图格式。
wx.CURHandle	wx.BITMAP_TYPE_CUR	Windows 光标 图标格式。
wx.GIFHandler	wx.BITMAP_TYPE_GIF	图形交换格式。由于版权限制，这个处理器不保存图像。
wx.ICOHandler	wx.BITMAP_TYPE_ICO	Windows 图标格式。
wx.IFFHandler	wx.BITMAP_TYPE_IFF	交换文件格式。这个处理器只载入图像，它不保存它们。
wx.JPEGHandler	wx.BITMAP_TYPE_JPEG	联合图形专家组格式。
wx.PCXHandler	wx.BITMAP_TYPE_PCX	PC 画刷格式。当以这种格式保存时， wxPython 计算在这个图像中的不同颜色的数量。如果可能的话，这个图像被保存为一个8位图像（也就是说，如果它有256种或更少的颜色）。否则它保存为24位。
wx.PNGHandler	wx.BITMAP_TYPE_PNG	便携式网络图形格式。
wx.PNMHandler	wx.BITMAP_TYPE_PNM	只能载入 ASCII 或原始的 RGB 图像。图像被该处理器保存为原始的 RGB 。
wx.TIFFHandler	wx.BITMAP_TYPE_TIF	标签图像文件格式。
wx.XPMHandler	wx.BITMAP_TYPE_XPM	XPixMap 格式。
自动	wx.BITMAP_TYPE_ANY	自动检测使用的格式，然后调用相应的处理器。

要使用一个 MIME 类型来标识文件，而不是一个处理器类型 ID 的话，请使用函数 wx.ImageFromMime(name, mimetype, index =-1)，其中的 name 是文件名， mimetype 是有关文件类型的字符串。参数 index 表示在图像文件包含多个图像的情况下要载入的图像的索引。这仅仅被 GIF, ICO, 和 TIFF 处理器使用。默认值-1表示选择默认的图像，被 GIF 和 TIFF 处理顺解释为每一个图像（ index =0），被 ICO 处理器解释为最大且最多色彩的一个。

创建 **image** （图像）对象

wxPython 使用不同的全局函数来创建不同种类的 wx.Image 对象。要创建一个有着特定尺寸的空图像，使用函数 wx.EmptyImage(width,height) ——在这个被创建的图像中所有的像素都是黑色。要创建从一个打开的流或 Python 文件类对象创建一个图像，使用 wx.ImageFromStream(stream,type = wx.BITMAP_TYPE_ANY, index =-1)。有时，根据一个原始的 RGB 数据来创建一个图像是有用的，这使用 wx.ImageFromData(width,height,data) ， data 是一个字符串，每套连续的三个字符代表一个像素的红，绿，蓝的组分。这个字符串的大小应该是 width *height* 3。

创建 **bitmap** （位图）对象

有几个方法可以创建一个位图对象。其中最基本的 wx.Bitmap 构造函数是 wx.Bitmap(name, type = wx.BITMAP_TYPE_ANY) 。参数 name 是一个文件名， type 可以是表12.1中的一个。如果 bitmap 类能够本地化地处理这个文件格式，那么它就处理，否则这个图像将自动地经由 wx.Image 载入并被转换为一个 wx.Bitmap 实例。

你可以使用方法 wx.EmptyBitmap(width,height,depth =-1)来创建一个空的位图——参数 width 和 height 是位图的尺度， depth 是结果图像的颜色深度。有两个函数使你能够根据原始的数据来创建一个位图。函数 wx.BitmapFromBits(bits, width, height, depth =-1)创建一个位图，参数 bits 是一个 Python 字节列表。这个函数的行为依赖于平台。在大多数平台上， bits 要么是1要么是0，并且这个函数创建一个单色的位图。在 Windows 平台上，数据被直接传递给 Windows 的 API 函数 CreateBitmap() 。函数 wxBitmapFromXPMData(listOfStrings) 一个 Python 字符串列表作为一个参数，以 XPM 格式读该字符串。

通过使用 wx.Bitmap 的构造函数 wx.BitmapFromImage(image, depth =-1)，你可以将一个图像转换为一个位图。参数 image 是一个实际 wx.Image 对象， depth 是结果位图的颜色深度。如果这个深度没有指定，那么使用当前显示器的颜色深度。你可以使用函数 wx.ImageFromBitmap(bitmap) 将位图转回为一个图像，通过传递一个实际的 wx.Bitmap 对象。在例12.1中，位图对象的创建使用了位图的构造函数，然后被用于构建 wx.StaticBitmap 窗口部件，这使得它们能够像别的 wxPython 项目一样被放入一个容器部件中。

我们能够对图像作些什么？

一旦你在 `wxPython` 中使用了图像，你就可以使用许多有用的方法来处理它，并且可以写一些强大的图像处理脚本。

你可以使用 `GetWidth()` 和 `GetHeight()` 方法来查询图像的尺寸。你也可以使用方法 `GetRed(x, y)`, `GetGreen(x, y)`, 和 `GetBlue(x, y)` 方法得到任意象素点的颜色值。这些颜色方法的返回值是一个位于0~255之间的整数（用C的术语，它是一个无符号整数，但是这个区别在 `Python` 中没有多大的意义）。同样，你能够使用 `SetRGB(x, y, red, green, blue)` 来设置一个像素点的颜色，其中的x和y是这个像素点的坐标，颜色的取值位于0~255之间。

你可以使用 `GetData()` 方法得到一大块区域中的所有数据。 `GetData()` 方法的返回值是一个大的字符串，其中的每个字符代表一个 `RGB` 元组，并且每个字符都可被认为是一个0~255之间整数值。这些值是有顺序的，第一个是位于像素点(0,0)的红色值，接下来的是位于像素点(0,0)的绿色值，然后是位于像素点(0,0)的蓝色值。再接下来的三个是像素点(0,1)的颜色值，如此等等。这个算法可以使用下面的 `Python` 伪代码来定义：

```
def GetData(self):
result = ""
for y in range(self.GetHeight()):
for x in range(self.GetWidth()):
result.append(chr(self.GetRed(x,y)))
result.append(chr(self.GetGreen(x,y)))
result.append(chr(self.GetBlue(x,y)))
return result
```

当使用对应的 `SetData(data)` 方法读取类似格式的 `RGB` 字符串值时，有两件事需要知道。第一， `SetData(data)` 方法不执行范围或边界检查，以确定你读入的字符串的值是否在正确的范围内或它的长度是否是给定图像的尺寸。如果你的值不正确，那么该行为是未定义的。第二，由于底层是C++代码管理内在，所以将 `GetData()` 的返回值传递给 `SetData()` 是一个坏的方法——你应该构造一个新的字符串。

图像数据字符串可以很容易地与别的 `Python` 类型作相互的转换，这使得很容易以整数的形式来访问和处理它，诸如数组或数字类型。例如，如果你太久的注视一个东西会损伤眼睛一样，试试这样：

```
import array
img = wx.EmptyImage(100,100)
a = array.array('B', img.GetData())
for i in range(len(a)):
a[i] = (25+i) % 256
img.SetData(a.tostring())
```

表12.2定义了一些 `wx.Image` 的方法，这些方法执行简单的图像处理。

这些方法只是图像处理的开始部分。在接下来的部分，我们将给你展示两个方法，它们处理透明和半透明图像这一更复杂的主题。

表12.2 `wx.Image` 的图像处理方法

`ConvertToMono(r, g, b)` ：返回一个与原尺寸一致的 `wx.Image` ，其中所有颜色值为(`r, g, b)` 的像素颜色改为白色，其余为黑色。原图像未改变。

`Mirror(horizontally = True)` ：返回原图像的一个镜像图像。如果 `horizontally` 参数是 `True` ，那么镜像图像是水平翻转了的，否则是垂直翻转了的。原图像没有改变。

`Replace(r1, g1, b1, r2, g2, b2)` ：改变调用该方法的图像的所有颜色值为 `r1, g1, b1` 的像素的颜色为 `r2, g2, b2` 。

`Rescale(width, height)` ：改变图像的尺寸为新的宽度和高度。原图像也作了改变，并且颜色按比例地调整到新的尺寸。

`Rotate(angle, rotationCentre, interpolating = True, offestAfterRota = None)` ：返回旋转原图像后的一个新的图像。参数 `angle` 是一个浮点数，代表所转的弧度。 `rotationCentre` 是一个 `wx.Point` ，代表旋转的中心。如果 `interpolating` 为 `True` ，那么一个较慢而精确的算法被使用。 `offsetAfterRotation` 是一个坐标点，表明在旋转后图像应该移位多少。任何未被覆盖的空白像素将被设置为黑色，或如果该图像有一个遮罩色，设置为遮罩色（ `mask color` ）。

`Rotate90(clockwise = True)` ：按照参数 `clockwise` 的布尔值，控制图像按顺或逆时针方向作90度的旋转。

`Scale(width, height)` ：返回一个原图像的拷贝，并按比例改变为新的宽度和高度。

设置图像的遮罩以指定一个透明的图像

图像遮罩是图像中的一个特殊的颜色集，当图像显示在其它显示部分之上时，它扮演透明度的角色。你可以使用 `SetMaskColor(red, green, blue)` 方法来设置一个图像遮罩，其中的 `red, green, blue` 定义图像遮罩的颜色。如果你想关闭遮罩，可以使用 `SetMask(False)` ，重置使用 `SetMask(True)` 。方法 `HasMask()` 返回与当前遮罩状态相关的一个布尔值。你也可以使用方法 `SetMaskFromImage(mask, mr, mg, mb)` 根据同一尺寸的另一图像设置遮罩——在这种情况下，遮罩被定义为在遮罩 `wx.Image` 中有着颜色 `mr, mg, mb` 的所有像素，而不管在主图像中那些像素是什么颜色。这使得你在创建一个遮罩中有了很大的灵活性，因为你不必再担心在你原图像中的像素的颜色。你可以使用 `GetMaskRed()` ， `GetMaskGreen(),` 和 `GetMaskBlue()` 获取遮罩色。如果一个有遮罩的图像被转换为一个 `wx.Bitmap` ，那么遮罩被自动转换为一个 `wx.Mask` 对象并赋给该位图。

设置 **`alpha`** 值来指定一个透明的图像

`alpha` 值是指定一个透明或部分透明图像的另一个方法。每个像素都有一个 `alpha` 值，取值位于0（如果图像在该像素是完全透明的）到255（如果图像在该像素点是完全不透明的）之间。你可以使用 `SetAlphaData(data)` 方法来设置 `alpha` 值，它要求类似于 `SetData()` 的字符串字节值，但是每个像素只有一个值。和 `SetData()` 一样， `SetAlphaData()` 不进行范围检查。你可以使用 `HasAlpha()` 来看是否设置了 `alpha` 值，你也可以使用 `GetAlphaData()` 来得到全部的数据集。你也可以使用 `SetAlpha(x, y, alpha)` 来设定一个特定的像素的 `alpha` 值，并使用 `GetAlpha(x, y)` 来得到该值。

与 `wx.Image` 的图像处理功能相对照， `wx.Bitmap` 的相对少些。几乎所有 `wx.Bitmap` 的方法都是简单得得到诸如宽度、高度和颜色深度这类的属性。

第三部分 高级 `wxPython`

在这一部分，我们以三个更复杂的窗口部件对象作为开始，并转到并不是每个 `wxPython` 程序都有的特性上来。

在第13章“建造列表控件并管理项目”中，我们涉及到了列表控件。较简单的列表框更为高级的是，这个列表控件可以用来产生一个十分类似于 `Windows` 资源管理器的东西，包括不同的模式。你将看到如何在模式间进行切换，添加文本和图像到列表，还有响应用户事件。

14章“协调 `grid` 控件”为列表增加了 `grid` 控件。 `grid` 是十分灵活的，我们将给你展示管理网格中数据的所有方法，以及自定义网格显示和编辑的机制。

第15章“树形控件”处理树形控件，它使你能够显示树的层次。我们将展示如何管理树的数据，遍历树和自定义树的显示。

在第16章“在你的应用程序中加入 `HTML` ”中，我们将展示用 `HTML` 来指定文本标签和打印的样式是多么的便利。我们将给你展示 `HTML` 窗口部件是如何工作的，以及它们关于标准 `HTML` 的局限性。

第17章“ `wxPython` 的打印框架”涉及到了打印，如何绘制到一个打印机，以及如何管理在 `wxPython` 和底层打印系统间通信的标准打印对话框。我们也将展示如何添加打印预览功能。

第18章“使用 `wxPython` 的其它功能”所涉及的东西不是无论哪都适用。这章涉及了通过剪贴板传递数据以及如何管理拖放操作等方面。我们还将展示如何使用计时器(`timer`) 创建定时的行为，并提供少量在 `wxPython` 应用程序中关于线程方面的想法。

第十三章 建造列表控件并管理列表项

本章内容：

- 创建不同样式的列表控件
- 处理列表中的项目
- 响应列表中用户的选择
- 编辑标签和对列表排序
- 创建大的列表控件

在 `wxPython` 中，有两个控件，你可以用来显示基于列表的信息。较简单的是列表框，它是可滚动的单列列表，类似于使用 `HTML` 的 `select` 标记所得到的。列表框已经在第8章中讨论过了，本章不再作进一步的讨论。

本章讨论较为复杂的列表：列表控件，它是一个完整特性的列表窗口部件。这个列表控件可以每行显示多列信息，并可基于任一行进行排序，还能以不同的样式显示。对于列表控件的每个部分的细节显示，你有很大的灵活性。

建造一个列表控件

列表控件能够以下面四种不同模式建造：

- 图标(`icon)`
- 小图标(`small icon)`
- 列表(`list)`
- 报告(`report)`

如果你用过微软 `Windows` 的资源管理器(`Explorer)` 或 `Mac` 的 `Finder` ，那么这些方式你应该熟悉。我们将通过说明如何建立四种不同模式的列表作为开始来介绍列表控件。

什么是图标模式？

列表控件看起来类似于微软的 `Windows` 资源管理器的一个文件树系统的显示面板。它以四种模式之一的一种显示一个信息的列表。默认模式是图标模式，显示在列表中的每个元素都是一个其下带有文本的一个图标。图13.1显示了一个图标模式的列表。

例13.1是产生图13.1的代码。注意这个例子使用了同目录下的一些. `png` 文件。

例**13.1** 创建一个图标模式的列表

```
#-*- encoding:UTF-8 -*-
import wx
import sys, glob

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "wx.ListCtrl in wx.LC_ICON mode",
                          size=(600,400))

        # load some images into an image list
        il = wx.ImageList(32,32, True)#创建图像列表
        for name in glob.glob("icon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)

        # create the list control
#创建列表窗口部件
self.list = wx.ListCtrl(self, -1,
                style=wx.LC_ICON | wx.LC_AUTOARRANGE)

        # assign the image list to it
        self.list.AssignImageList(il, wx.IMAGE_LIST_NORMAL)

        # create some items for the list
#为列表创建一些项目
        for x in range(25):
            img = x % (il_max+1)
self.list.InsertImageStringItem(x,
                    "This is item %02d" % x, img)

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

图**13.1**

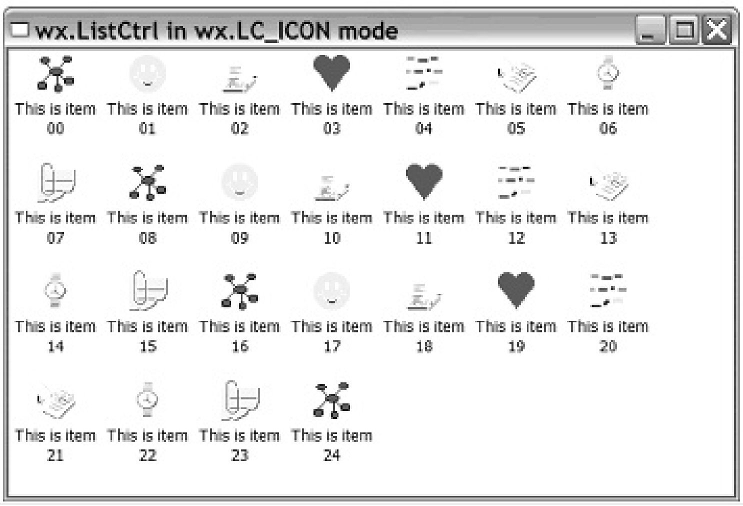


Figure 13.1
A sample list in icon mode

在例13.1中，`DemoFrame`创建了一个“`image list`（图像列表）”来包含对要显示的图像的引用，然后它建造并扩充了这个列表控件。我们将在本章稍后的部分讨论“`image list`（图像列表）”。

什么是小图标模式？

小图标模式类似标准的图标模式，但是图标更小点。图13.2以小图标模式显示了相同的列表。

当你想在窗口部件中放入更多的显示项目时，小图标模式是最有用的，尤其是当图标不够精细时。

图**13.2**

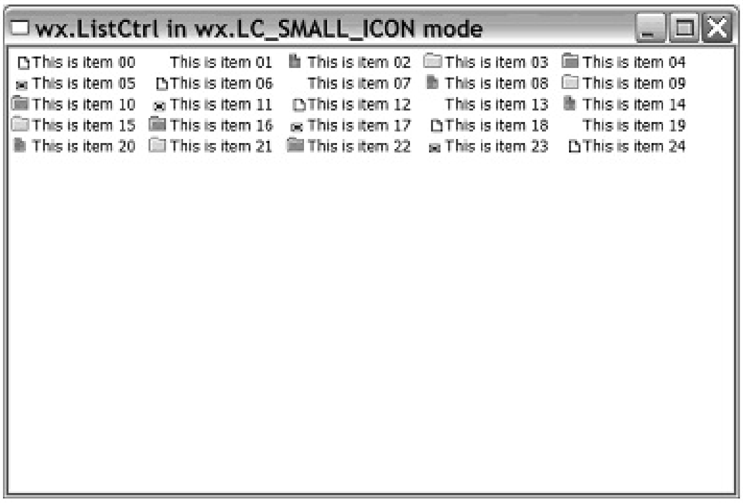


Figure 13.2
A sample list control in small icon mode

产生图13.2的示例代码如下：

```
import wx
import sys, glob

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "wx.ListCtrl in wx.LC_SMALL_ICON mode"
,
                          size=(600,400))

        # load some images into an image list
        il = wx.ImageList(16,16, True)
        for name in glob.glob("smicon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)

        # create the list control
        self.list = wx.ListCtrl(self, -1,
                                style=wx.LC_SMALL_ICON
                                | wx.LC_AUTOARRANGE
                                )

        # assign the image list to it
        self.list.AssignImageList(il, wx.IMAGE_LIST_SMALL)

        # create some items for the list
        for x in range(25):
            img = x % (il_max+1)
            self.list.InsertImageStringItem(x,
                                            "This is item %02d"
% x,
                                            img)

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

什么是列表模式？

在列表模式中，列表以多列的形式显示，一列到达底部后自动从下一列的上部继续，如图13.3所示。

列模式在相同元素的情况下，几乎与小图标模式所能容纳的项目数相同。对这两个模式的选择，主要是根据你的数据是按列组织好呢还是按行组织好。

图**13.3**

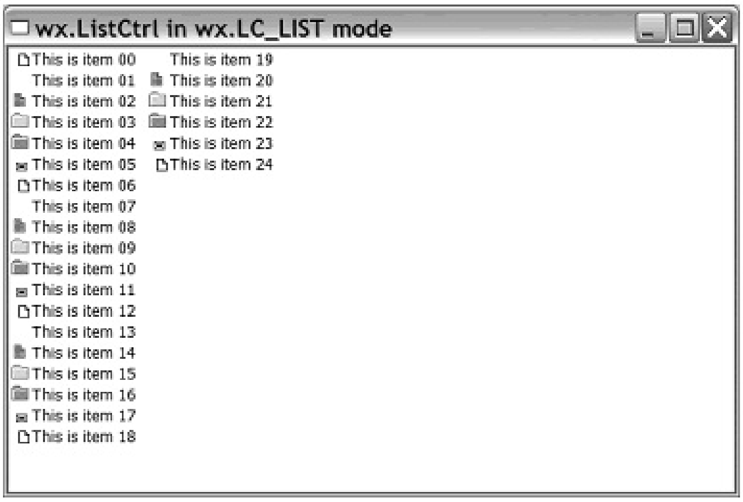


Figure 13.3
A sample list control in list mode

产生图13.3的示例代码如下：

```
import wx
import sys, glob

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "wx.ListCtrl in wx.LC_LIST mode",
                          size=(600,400))

        # load some images into an image list
        il = wx.ImageList(16,16, True)
        for name in glob.glob("smicon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)

        # create the list control
        self.list = wx.ListCtrl(self, -1, style=wx.LC_LIST)

        # assign the image list to it
        self.list.AssignImageList(il, wx.IMAGE_LIST_SMALL)

        # create some items for the list
        for x in range(25):
            img = x % (il_max+1)
            self.list.InsertImageStringItem(x,
                                            "This is item %02d"
% x,
                                            img)

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

在报告模式中，列表显示为真正的多列格式，每行可以有任一数量的列，如图13.4所示。

图13.4

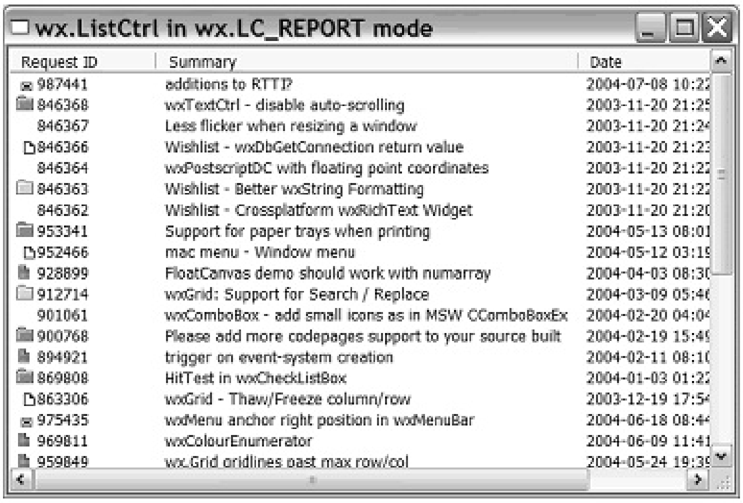


Figure 13.4
A sample list control in report mode

报告模式与图标模式不尽相同。例13.2显示了图13.4的代码。

例**13.2** 创建报告模式的一个列表

```
#!/usr/bin/python
#-*- encoding:UTF-8 -*-
import wx
import sys, glob, random
import data

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "wx.ListCtrl in wx.LC_REPORT mode",
                          size=(600,400))

        il = wx.ImageList(16,16, True)
        for name in glob.glob("smicon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)
        self.list = wx.ListCtrl(self, -1, style=wx.LC_REPORT)#创
建列表
        self.list.AssignImageList(il, wx.IMAGE_LIST_SMALL)

        # Add some columns
        for col, text in enumerate(data.columns):#增加列
            self.list.InsertColumn(col, text)

        # add the rows
        for item in data.rows:#增加行
            index = self.list.InsertStringItem(sys.maxint, item[
0])
            for col, text in enumerate(item[1:]):
                self.list.SetStringItem(index, col+1, text)

            # give each item a random image
            img = random.randint(0, il_max)
            self.list.SetItemImage(index, img, img)

        # set the width of the columns in various ways
        self.list.SetColumnWidth(0, 120)#设置列的宽度
        self.list.SetColumnWidth(1, wx.LIST_AUTOSIZE)
        self.list.SetColumnWidth(2, wx.LIST_AUTOSIZE)
        self.list.SetColumnWidth(3, wx.LIST_AUTOSIZE_USEHEADER)

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

注意：如果代码中有中文或中文注释，那么请在代码开头加上#--`encoding:UTF`*-8 --*

在接下来的部分，我们将讨论如何将值插入适当的位置。报告控件是最适合用于那些包含一两个附加的数据列的简单列表，它的显示逻辑没有打算做得很复杂。如果你的列表控件复杂的话，或包含更多的数据的话，那么建议你使用`grid`控件，说明见第14章。

如何创建一个列表控件？

一个`wxPython`列表控件是类`wx.ListCtrl`的一个实例。它的构造函数与其它的窗口部件的构造函数相似：

`wx.ListCtrl(parent, id, pos=wx.DefaultPosition,`

- `size=wx.DefaultSize, style=wx.LC_ICON, validator=wx.DefaultValidator, name=`" `listCtrl` ")

这些参数我们在其它的窗口部件的构造函数中见过。参数`parent`是容器部件，`id`是`wxPython`标识符，使用-1表明自动创建标识符。具体的布局由参数`pos`和`size`来管理。`style`控制模式和其它的显示方案——贯穿本章，我们都将看到这些值。参数`validator`用于验证特定的输入，我们在第9章讨论过。参数`name`我们很少使用。

样式（`style`）标记是一个位掩码，它管理列表控件的一些不同的特定。样式标记的第一组值用于设置列表的显示模式。默认模式是`wx.LC_ICON`。表13.1显示了列表控件的模式值。

表**13.1** 列表控件模式值

`wx.LC_ICON`	图标模式，使用大图标
`wx.LC_LIST`	列表模式
`wx.LC_REPORT`	报告模式
`wx.LC_SMALL_ICON`	图标模式，使用小图标

在图标或小图标列表中，有三个样式标记用来控件图标相对于列表对齐的。默认值是`wx.LC_ALIGN_TOP`，它按列表的顶部对齐。要左对齐的话，使用`wx.LC_ALIGN_LEFT`。样式`LC_AUTOARRANGE`使得当图标排列到达窗口右或底边时自动换行或换列。

表13.2显示了作用于报告列表显示的样式。

表**13.2** 报告列表的显示样式

`wx.LC_HRULES`	在列表的行与行间显示网格线（水平分隔线）
`wx.LC_NO_HEADER`	不显示列标题
`wx.LC_VRULES`	显示列与列之间的网格线（竖直分隔线）

样式标记可以通过位运算符来组合。使用 `wx.LC_REPORT | wx.LC_HRULES | wx.LC_VRULES` 组合可以得到一个非常像网格的一个列表。默认情况下，所有的列表控件都允许多选。要使得一次只能选列表中的一个项目，可以使用标记 `wx.LC_SINGLE_SEL` 。

与我们见过的其它的窗口部件不同，列表控件增加了一对用于在运行时改变已存在的列表控件的样式标记的方法。 `SetSingleStyle(style, add=True)` 方法使你能够增加或去掉一个样式标记，这依赖于参数 `add` 的值。 `listCtrl.SetSingleStyle(LC_HRULES,True)` 将增加水平分隔线，而 `listCtrl.SetSingleStyle(LC_HRULES,False)` 将去掉水平分隔线。 `listCtrl` 代表具体的列表控件。 `SetWindowStyleFlag(style)` 能够重置整个窗口的样式，如 `SetWindowStyleFlag(LC_REPORT | LC_NO_HEADER)` 。这些方法对于在运行时修改列表控件的样式就有用处的。

处理列表中的项目

一旦列表控件被创建，你就能够开始将信息添加到该列表中。在 `wxPython` 中，对于纯文本信息和对与列表中的每个项目相关的图像的处理是不同的。在接下来的几节里，我们将如何添加图像和文本到你的列表控件中。

什么是一个图像列表以及如何将图像添加给它？

在我们讨论信息是如何被添加到列表控件之前，我们需要对列表如何控制图像说两句。任何使用在一个列表控件中的图像，首先必须被添加到一个图像列表，图像列表是一个图像索引数组，使用列表控件存储。当一个图像与列表中的一个特定项目相关联时，图像列表中的该图像的索引被用来引用该图像，而非使用图像本身。该机制确保每个图像只被装载一次。这是为了在一个图标被列表中的几个项目重复使用时节约内存。它也允许相同图像的多个版本之间的相对直接的连接，这些版本被用来表示不同的模式。关于创建 `wxPython` 图像和位图的更多的信息，请看第12章。

创建一个图像列表

图像列表是 `wx.ImageList` 的一个实例，构造函数如下：

```
wx.ImageList(width, height, mask=True, initialCount=1)
```

参数 `width` 和 `height` 指定了添加到列表中的图像的像素尺寸。比指定大小大的图像是不允许的。参数 `mask` 是一个布尔值。如果为 `True` ，假如图像有遮罩，则使用遮罩绘制图像。参数 `initialCount` 设置列表的初始的内在尺寸。如果你知道列表会很大，那么指定初始量可以获得更多的内存分配以便稍后使用。

添加及移去图像

你可以使用方法 `Add(bitmap, mask=wx.NullBitmap)` 来将一个图像添加到列表，参数 `bitmap` 和 `mask` 都是 `wx.Bitmap` 的实例。 `mask` 参数是一个单色位图，它代表该图像的透明部分，如果指定了 `mask` 参数的话。如果位图已经有一个与之相关的遮罩，那么该遮罩被默认使用。如果位图没有一个遮罩，并且你不使用

单色透明映射，但设置了该位图的一个特定颜色作为这个透明色的话，那么你可以使用 `AddWithColourMask(bitmap` ， `colour)` 方法，其中参数 `colour` 是用作遮罩的 `wxPython` 颜色（或它的颜色名）。如果你有一个 `wx.Icon` 对象要添加到图像列表，可以使用方法 `AddIcon(icon)` 。所有这些添加方法都返回这个新加的图像在列表中的索引值，你可以保留索引值以便日后使用该图像。

下面的代码片断显示了一个创建图像列表的例子（类似于例13.1中的）。

```
il = wx.ImageList(32, 32, True)
for name in glob.glob("icon??.png"):
bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
il_max = il.Add(bmp)
```

然后这个图像列表必须被赋给一个列表控件，使用下面的方法：

`self.list.AssignImageList(il, wx.IMAGE_LIST_NORMAL)`

要从图像列表删除一个图像，可以使用 `Remove(index)` 方法，其中的 `index` 是图像在图像列表中的整数索引值。这个方法会修删除点之后的图像在图像列表中的索引值，如果在你的程序中有对特定的索引存在依赖关系的话，这可能会导致一些问题。要删除整个图像列表，使用 `RemoveAll()` 。你可以使用方法 `Replace(index, bitmap, mask=wx.NullBitmap)` 修改特定索引相关的位图，其中 `index` 是列表中要修改处的索引， `bitmap` 和 `mask` 与 `Add()` 方法中的一样。如果要修改的项目是一个图标，可以使用方法 `ReplaceIcon(index, icon)` 。这里没有处理颜色遮罩的替换方法。

使用图像列表

通过使用方法 `GetImageCount()` ，你能够得到图像列表的长度，使用 `GetSize()` 方法，你可以得到其中个个图像的尺寸，它返回一个(`width, height)` 元组。

在列表控件上下文中没有直接相关的图像的时候，你也可以根据图像列表绘制一个图像到设备上下文中。关于设备上下文的更多信息，请看第6章和第12章。这个方法是 `Draw` ，如下所示：

```
Draw(index, dc, x, y, flags=wx.IMAGELIST_DRAW_NORMAL,
        solidBackground=False)
```

在这个调用中，参数 `index` 是要绘制的项目在图像列表中的索引，参数 `dc` 是要绘制到的一个 `wx.DC` 设备上下文。 `flags` 控制图像被如何绘制， `flags` 的可取值
有 `wx.IMAGELIST_DRAW_NORMAL, wx.IMAGELIST_DRAW_TRANSPARENT, wx.IMAGEL`
和 `wx.IMAGELIST_DRAW_FOCUSED` 。如果 `solidBackground` 为 `True` ，那么该绘制方法使用一个更快的算法工作。

一旦你有了一个图像列表，你就需要将它附给列表控件。这个以通过后面的任一个方法来实现：`AssignImage(imageList, which)` 或 `SetImage(imageList, which)` 。 `i` 参数是一个图像列表，参数 `which` 是标记值： `wx.IMAGE_LIST_NORMAL` 或 `wx.IMAGE_LIST_SMALL` 。这两个方法的唯一的不同之处是C++对图像列表的处理方面。对于 `AssignImage()` ，图像列表变成了列表控件的一部分，并随列表控件的销毁而销毁。对于 `SetImage()` ，图像列表有自己的生命周期，当列表控件被销毁时不被自动处理，只是当其 `Python` 对象退出作用域时，才被处理。

可以赋给列表控件两个图像列表。普通的图像列表（使用了 `wx.IMAGE_LIST_NORMAL` ）被用于标准的图标模式。小图像列表（使用了 `wx.IMAGE_LIST_SMALL` ）被用于报告和小图标模式。在大多数情况下，你只需要一个图像列表，但是如果你希望列表以多模式显示（这样用户可以从普通模式切换到小图标模式），那么你应该两个都提供。如果你这样做了，那么记住，列表控件中的选项将只会经由图像列表中的索引知道相关的图像。如果文档图标在普通尺寸的图像列表中有两个索引，那么也必须在小图像列表中有两个索引。

关于列表控件还有一个相关的 `get` *方法： `GetImageList(which)` ，它返回与 `which` 标记参数相关的图像列表。

如何对一个列表添加或删除项目？

在你能显示一个列表之前，你需要给它增加文本信息。在一个图标列表中，你可以增加新的项目如图标、字符串或两个都添加。在一个报告视图中，你也可以在设置了初始图标和/或字符串后，为一行中的不同的列设置信息。用于处理列表控件项目的方法的 `API` 及其命名习惯与迄今为止我们所见过的其它一些控件的是有区别的，因此，尽管你已经理解了菜单或列表框是如何工作的，但是你仍将需要读这一节。

对于一个图标列表，增加文本信息到列表控件是一个单步的处理过程，但是对于一个报告列表就需要多步才行。通常对于每个列表，第一步是在行中增加第一个项目。对于报告列表，你必须分别地增加列和列中的信息，而非最左边的一个。

增加一个新行

要增加一个新行，使用 `InsertItem()` 这类的一种方法。具体所用的方法依赖于你所插入的项目的类型。如果你仅仅插入一个字符串到列表中，使用 `InsertStringItem(index, label)` ，其中的 `index` 是要插入并显示新项目的行的索引。如果你只插入一个图像，那么使用 `InsertImageItem(index, imageIndex)` 。在这种情况下，这 `index` 是要插入图像的行的索引， `imageIndex` 是附加到该列表控件的图像列表中的图像的索引。要插入一个图像项目，图像列表必须已经被创建并赋值。如果你使用的图像索引超出了图像列表的边界，那么你将得到一个空图像。如果你想增加一个既有图像又有字符串标签的项目，使用 `InsertImageStringItem(index, label, imageIndex)` 。这个方法综合了前面两个方法的参数，参数的意义不变。

在内部，列表控件使用类`wx.ListItem`的实例来管理有关它的项目的信息。我还要说的是，最后一种插入项目到列表控件中方法是`InsertItem(index, item)`，其中的`item`是`wx.ListItem`的一个实例。对于`wx.ListItem`，这里我们不将做很详细的说明，这是因为你几乎不会用到它并且该类也不很复杂——它几乎都是由`get`和*`set`*方法组成的。一个列表项的几乎所有属性都可通过列表控件的方法来访问。

增加列

要增加报告模式的列表控件的列，先要创建列，然后设置每行/列对的单独的数据单元格。使用`InsertColumn()`方法创建列，它的语法如下：

```
InsertColumn(col, heading, format=wx.LIST_FORMAT_LEFT, width=-1)
```

在这个方法中，参数`col`是列表中的新列的索引，你必须提供这个值。参数`heading`是列标题。参数`format`控件列中文本的对齐方式，取值有：`wx.LIST_FORMAT_CENTRE`、`wx.LIST_FORMAT_LEFT`、和`wx.LIST_FORMAT_RIGHT`。参数`width`是列的初始显示宽度（像素单位）——用户可以通过拖动列的头部的边来改变它的宽度。要使用一个`wx.ListItem`对象来设置列的话，也有一个名为`InsertColumnInfo(info)`的方法，它要求一个列表项作为参数。

设置多列列表中的值

你可能已经注意到使用前面说明的行的方法来插入项目，对于一个多列的报告列表来说只能设置最初的那列。要在另外的列中设置字符串，可以使用方法`SetStringItem()`。

```
SetStringItem(index, col, label, imageId=-1)
```

参数`index`和`col`是你要设置的单元格的行和列的索引。你可以设定`col`为0来设置第一列，但是参数`index`必须对应列表控件中已有的行——换句话说，这个方法只能对已有的行使用。参数`label`是显示在单元格中文本，参数`imageId`是图像列表中的索引（如果你想在单元格中显示一个图像的话可以设置这个参数）。

`SetStringItem()`是`SetItem(info)`方法的一种特殊情况，`SetItem(info)`方法要求一个`wx.ListItem`实例。要使用这个方法，在将`wx.ListItem`实例增加到一个列表之前，要先设置它行，列和其它的参数。你也可以使用`GetItem(index,col=0)`方法来得到单元格处的`wx.ListItem`实例，默认情况下，该方法返回一行的第一列，你可以通过设置参数`col`来选择其它列的一项。

项目属性

有许多的 `get` 和 *`set`* 方法使你能够指定部分项目。通常这些方法工作在一行的第一列上。要得工作在其它的列上，你需要使用 `GetItem()` 来得到项目，并使用项目类的 `get` 和 *`set`* 方法。你可以使用 `SetItemImage(item, image, selImage)` 来为一个项目设置图像，其中的 `item` 参数是该项目在列表中的索引， `image` 和 `selImage` 都是图像列表中的索引，分别代表通常显示的图像和被选中时显示的图像。你可以通过使用 `GetItemText(item)` 和 `SetItemText(item,text)` 方法来得到或设置一个项目的文本。

你可以使用 `GetItemState(item,stateMask)` 和 `SetItemState(item, state, stateMas` 来得到或设置单独一个项目的状态。 `state` 和 `stateMask` 的取值见表13.3。参数 `state` （及 `GetItemState` 的返回值）是项目的实际状态， `stateMask` 是当前关注的所有可能值的一个掩码。

你可以使用 `GetColumn(col)` 来得到一个指定的列，它返回索引 `col` 处的列的 `wx.ListItem` 实例。

表**13.3** 状态掩码参数

状态及说明如下：

`wx.LIST_STATE_CUT`	被剪切状态。这个状态只在微软 `Windows` 下有效。
`wx.LIST_STATE_DONTCARE`	无关状态。这个状态只在微软 Windows 下有效。
`wx.LIST_STATE_DropHILITED`	拖放状态。项目显示为高亮，这个状态只在微软 `Windows` 下有效。
`wx.LIST_STATE_FOCUSED`	获得光标焦点状态。
`wx.LIST_STATE_SelectED`	被选中状态。

你也可以用 `SetColumn(col, item)` 方法对一个已添加的列进行设置。你也可以在程序中用 `GetColumnWidth(col)` 方法方法得到一个列的宽度，该方法返回列表的宽度（像素单位）——显然这只对报告模式的列表有用。你可以使用 `SetColumnWidth(col,width)` 来设置列的宽度。这个 `width` 可以是一个整数值或特殊值，这些特殊值有： `wx.LIST_AUTOSIZE` ，它将列的宽度设置为最长项目的宽度，或 `wx.LIST_AUTOSIZE_USEHEADER` ，它将宽度设置为列的首部文本（列标题）的宽度。在非 `Windows` 操作系统下， `wx.LIST_AUTOSIZE_USEHEADER` 可能只自动地将列宽度设置到80像素。

如果你对已有的索引不清楚了，你可以查询列表中项目的数量。方法有 `GetColumnCount()` ，它返回列表中所定义的列的数量， `GetItemCount()` 返回行的数量。如果你的列表是列表模式，那么方法 `GetCountPerPage()` 返回每列中项目的数量。

要从列表中删除项目，使用 `DeleteItem(item)` 方法，参数 `item` 是项目在列表中的索引。如果你想一次删除所有的项目，可以使用 `DeleteAllItems()` 或 `ClearAll()` 。你可以使用 `DeleteColumn(col)` 删除一列， `col` 是列的索引。

响应用户

通常，一个列表控件在当用户选择了列表中的一个项目后都要做一些事情。在接下来的部分， 我们将展示一个列表控件都能响应哪些事件，并提供一个使用列表控件事件的例子。

如何响应用户在列表中的选择？

像别的控件一样，列表控件也触发事件以响应用户的动作。你可以像我们在第三章那样使用 `Bind()` 方法为这些 事件设置处理器。所有这些事件处理器都接受一个 `wx.ListEvent` 实例， `wx.ListEvent` 是 `wx.CommandEvent` 的子类。 `wx.ListEvent` 有少量专用的 `get` *方法。某些属性只对特定的事件类型有效，这些特定的事件类型在本章的另外部分说明。适用于所有事件类型的属性见表13.4。

表**13.4** **`wx.ListEvent`** 的属性

`GetData()`	与该事件的列表项相关的用户数据项
`GetKeyCode()`	在一个按键事件中，所按下的键的键码
`GetIndex()`	得到列表中与事件相关的项目的索引
`GetItem()`	得到与事件相关的实际的 `wx.ListItem`
`GetImage()`	得到与事件相关单元格中的图像
`GetMask()`	得到与事件相关单元格中的位掩码
`GetPoint()`	产生事件的实际的鼠标位置
`GetText()`	得到与事件相关的单元格中的文本

这儿有几个关于 `wx.ListEvent` 的不同的事件类型，每个都可以有一个不同的处理器。某些关联性更强的事件将在后面的部分讨论。表13.5列出了选择列表中的项目时的所有事件类型。

表**13.5** 与选择一个列表控件中的项目相关的事件类型

EVT_LIST_BEGIN_DRAG	当用户使用鼠标左按键开始一个拖动操作时，触发该事件
EVT_LIST_BEGIN_RDRAG	当用户使用鼠标右按键开始一个拖动操作时，触发该事件
EVT_LIST_Delete_ALL_ITEMS	调用列表的 DeleteAll() 将触发该事件
EVT_LIST_Delete_ITEM	调用列表的 Delete() 将触发该事件
EVT_LIST_Insert_ITEM	当一个项目被插入到列表中时，触发该事件
EVT_LIST_ITEM_ACTIVATED	用户通过在已选择的项目上按下回车或双击来激活一个项目时
EVT_LIST_ITEM_DESelectED	当项目被取消选择时触发该事件
EVT_LIST_ITEM_FOCUSED	当项目的焦点变化时触发该事件
EVT_LIST_ITEM_MIDDLE_CLICK	当在列表上敲击了鼠标的中间按钮时触发该事件
EVT_LIST_ITEM_RIGHT_CLICK	当在列表上敲击了鼠标的右按钮时触发该事件
EVT_LIST_ITEM_SelectED	当通过敲击鼠标左按钮来选择一个项目时，触发该事件
EVT_LIST_ITEM_KEY_DOWN	在列表控件已经获得了焦点时，一个按键被按下将触发该事件

下节中例13.3将提供一个关于上述事件中的一些事件的应用例子。

如何响应用户在一个列的首部中的选择？

除了用户在列表体中触发的事件以外，还有在报告列表控件的列首中所触发的事件。列事件创建的 wx.ListEvent 对象有另一个方法： GetColumn() ，该方法返回产生事件的列的索引。如果事件是一个列边框的拖动事件，那么这个索引是所拖动的边框的左边位置。如果事件是一个敲击所触发的，且敲击不在列内，那么该方法返回－1。表13.6包含了列事件类型的列表。

表**13.6** 列表控件列事件类型

EVT_LIST_COL_BEGIN_DRAG	当用户开始拖动一个列的边框时，触发该事件
EVT_LIST_COL_CLICK	列表首部内的一个敲击将触发该事件
EVT_LIST_COL_RIGHT_CLICK	列表首部内的一个右击将触发该事件
EVT_LiST_COL_END_DRAG	当用户完成对一个列表边框的拖动时，触发该事件

例13.3显示了一些列表事件的处理，并也提供了方法的一些演示。

例**13.3** 一些不同列表事件和属性的一个例子

```
import wx
import sys, glob, random
import data

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Other wx.ListCtrl Stuff",
                          size=(700,500))
        self.list = None
        self.editable = False
        self.MakeMenu()
        self.MakeListCtrl()

    def MakeListCtrl(self, otherflags=0):
        # if we already have a listctrl then get rid of it
        if self.list:
            self.list.Destroy()

        if self.editable:
            otherflags |= wx.LC_EDIT_LABELS

        # load some images into an image list
        il = wx.ImageList(16,16, True)
        for name in glob.glob("smicon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)

        # create the list control
        self.list = wx.ListCtrl(self, -1, style=wx.LC_REPORT|oth
erflags)

        # assign the image list to it
        self.list.AssignImageList(il, wx.IMAGE_LIST_SMALL)

        # Add some columns
        for col, text in enumerate(data.columns):
            self.list.InsertColumn(col, text)
```

```
        # add the rows
        for row, item in enumerate(data.rows):
            index = self.list.InsertStringItem(sys.maxint, item[
0])
            for col, text in enumerate(item[1:]):
                self.list.SetStringItem(index, col+1, text)

            # give each item a random image
            img = random.randint(0, il_max)
            self.list.SetItemImage(index, img, img)

            # set the data value for each item to be its positio
n in
            # the data list
            self.list.SetItemData(index, row)

        # set the width of the columns in various ways
        self.list.SetColumnWidth(0, 120)
        self.list.SetColumnWidth(1, wx.LIST_AUTOSIZE)
        self.list.SetColumnWidth(2, wx.LIST_AUTOSIZE)
        self.list.SetColumnWidth(3, wx.LIST_AUTOSIZE_USEHEADER)

        # bind some interesting events
        self.Bind(wx.EVT_LIST_ITEM_SelectED, self.OnItemSelected
, self.list)
        self.Bind(wx.EVT_LIST_ITEM_DESelectED, self.OnItemDesele
cted, self.list)
        self.Bind(wx.EVT_LIST_ITEM_ACTIVATED, self.OnItemActivat
ed, self.list)

        # in case we are recreating the list tickle the frame a
bit so
        # it will redo the layout
        self.SendSizeEvent()

    def MakeMenu(self):
        mbar = wx.MenuBar()
        menu = wx.Menu()
        item = menu.Append(-1, "E \tAlt-X")
        self.Bind(wx.EVT_MENU, self.OnExit, item)
        mbar.Append(menu, "  ")

        menu = wx.Menu()
        item = menu.Append(-1, "Sort ascending")
self.Bind(wx.EVT_MENU, self.OnSortAscending, item)
        item = menu.Append(-1, "Sort descending")
        self.Bind(wx.EVT_MENU, self.OnSortDescending, item)
        item = menu.Append(-1, "Sort by submitter")
        self.Bind(wx.EVT_MENU, self.OnSortBySubmitter, item)

        menu.AppendSeparator()
        item = menu.Append(-1, "Show selected")
```

```
self.Bind(wx.EVT_MENU, self.OnShowSelected, item)
        item = menu.Append(-1, "Select all")
        self.Bind(wx.EVT_MENU, self.OnSelectAll, item)
        item = menu.Append(-1, "Select none")
        self.Bind(wx.EVT_MENU, self.OnSelectNone, item)

        menu.AppendSeparator()
        item = menu.Append(-1, "Set item text colour")
        self.Bind(wx.EVT_MENU, self.OnSetTextColour, item)
        item = menu.Append(-1, "Set item background colour")
        self.Bind(wx.EVT_MENU, self.OnSetBGColour, item)

        menu.AppendSeparator()
        item = menu.Append(-1, "Enable item editing", kind=wx.IT
EM_CHECK)
        self.Bind(wx.EVT_MENU, self.OnEnableEditing, item)
        item = menu.Append(-1, "Edit current item")
        self.Bind(wx.EVT_MENU, self.OnEditItem, item)
        mbar.Append(menu, "  ")

        self.SetMenuBar(mbar)

    def OnExit(self, evt):
        self.Close()

    def OnItemSelected(self, evt):
        item = evt.GetItem()
        print "Item selected:", item.GetText()

    def OnItemDeselected(self, evt):
        item = evt.GetItem()
        print "Item deselected:", item.GetText()

def OnItemActivated(self, evt):
        item = evt.GetItem()
        print "Item activated:", item.GetText()

    def OnSortAscending(self, evt):
        # recreate the listctrl with a sort style
        self.MakeListCtrl(wx.LC_SORT_ASCENDING)

    def OnSortDescending(self, evt):
        # recreate the listctrl with a sort style
        self.MakeListCtrl(wx.LC_SORT_DESCENDING)

    def OnSortBySubmitter(self, evt):
        def compare_func(row1, row2):
            # compare the values in the 4th col of the data
            val1 = data.rows[row1][3]
            val2 = data.rows[row2][3]
            if val1   val2: return -1
            if val1   val2: return 1
            return 0
```

```
        self.list.SortItems(compare_func)

    def OnShowSelected(self, evt):
        print "These items are selected:"
        index = self.list.GetFirstSelected()
        if index == -1:
            print "\tNone"
            return
        while index != -1:
            item = self.list.GetItem(index)
            print "\t%s" % item.GetText()
            index = self.list.GetNextSelected(index)

    def OnSelectAll(self, evt):
        for index in range(self.list.GetItemCount()):
            self.list.Select(index, True)

    def OnSelectNone(self, evt):
        index = self.list.GetFirstSelected()
        while index != -1:
            self.list.Select(index, False)
            index = self.list.GetNextSelected(index)

    def OnSetTextColour(self, evt):
        dlg = wx.ColourDialog(self)
        if dlg.ShowModal() == wx.ID_OK:
            colour = dlg.GetColourData().GetColour()
            index = self.list.GetFirstSelected()
            while index != -1:
                self.list.SetItemTextColour(index, colour)
                index = self.list.GetNextSelected(index)
        dlg.Destroy()

    def OnSetBGColour(self, evt):
        dlg = wx.ColourDialog(self)
        if dlg.ShowModal() == wx.ID_OK:
            colour = dlg.GetColourData().GetColour()
            index = self.list.GetFirstSelected()
            while index != -1:
                self.list.SetItemBackgroundColour(index, colour)
                index = self.list.GetNextSelected(index)
        dlg.Destroy()

    def OnEnableEditing(self, evt):
        self.editable = evt.IsChecked()
        self.MakeListCtrl()

    def OnEditItem(self, evt):
        index = self.list.GetFirstSelected()
        if index != -1:
            self.list.EditLabel(index)
```

```
class DemoApp(wx.App):
    def OnInit(self):
        frame = DemoFrame()
        self.SetTopWindow(frame)
        print "Program output appears here..."
        frame.Show()
        return True

app = DemoApp(redirect=True)
app.MainLoop()
```

一旦你输入上面的代码并执行它，你将看到列表控件特性的演示，包括像项目的排序，这我们将在下一节讨论。

在这一节，我们将讨论对列表控件中的项目进行编辑、排序和想找。

如何编辑标签？

除了报告列表外，编辑一个列表中的项目是简单的，在报告列表中，用户只能编辑一行的第一个一。而对于其它的列表，则没有问题；每个项目的标准的标签都是可编辑的。

要使一个列表是可编辑的，则当列表被创建时要在构造函数中包含样?奖昙覔。

`list = wx.ListCtrl(self,` -1, `style=wx.LC_REPORT` | `wx.LC_EDIT_LABELS)`

如果这个编辑标记被设置了，那么用户就能够通过在一个已选择的列表项上敲击来开始一个编辑会话。编辑完后按下 `Enter` 键结束编辑会话，新的文本就变成了文本标签。在列表控件中的鼠标敲击也可结束编辑会话（一次只能有一个编辑会话）。按下 `Esc` 键则取消编辑会话，这样的话，新输入的文本就没有用了。

下面的两个事件类型是由编辑会话触发的。

- `EVT_LIST_BEGIN_LABEL_EDIT`
- `EVT_LIST_END_LABEL_EDIT`

记住，如果你想事件在被你的自定义的事件处理器处理后继续被处理，那么你需要在你的事件处理器中包括 `Skip()` 调用。当用户开始一个编辑会话时，一个 `EVT_LIST_BEGIN_LABEL_EDIT` 类型的列表事件被触发，当会话结束时（通过使用 `Enter` 或 `Esc` ），`EVT_LIST_END_LABEL_EDIT` 类型的列表事件被触发。你可以否决(`veto)` 编辑事件的开始，这样编辑会话就不会开始了。否决编辑事件的结束将阻止列表文本的改变。

`wx.ListEvent` 类有两个属性，这两个属性只在当处理一个 `EVT_LIST_END_LABEL_EDIT` 事件时才会用到。如果编辑结束并确认后， `GetLabel()` 返回列表项目标签的新文本，如果编辑被 `Esc` 键取消了，那么 `GetLabel()` 返回一个空字符串。这意味着你不能使用 `GetLabel()` 来区别“取

消”和“用户故意输入的空字符串标签”。如果必须的话，可以使用 `IsEditCancelled()` ，它在因取消而导致的编辑结束时返回 `True` ，否则返回 `False` 。

如果你想通过其它的用户事件来启动一个编辑会话的话，你可以在程序中使用列表控件的 `EditLabel(item)` 方法来触发一个编辑。其中的 `item` 参数是是要被编辑的列表项的索引。该方法触发 `EVT_LIST_BEGIN_LABEL_EDIT` 事件。

如果你愿意直接处理用于列表项编辑控件，你可以使用列表控件的方法 `GetEditControl()` 来得到该编辑控件。该方法返回用于当前编辑的文本控件。如果当前没有编辑，该方法返回 `None` 。目前该方法只工作于 `Windows` 操作系统下。

`boa` - `constructor` 简介

`boa` - `constructor` 是一个跨平台的 `Python` 集成开发环境和 `wxPython` 图形用户界面构建器。它提供了可视化方式的框架（窗口）的创建和处理、对象检视器(`object inspector)` 、编辑器、继承的等级、 `html` 文档字符串、高级的调试器和集成化的帮助系统。俨然一个用于 `Python` 的 `Delphi` 。

对的 `Zope` 支持：对象的创建和编辑。剪切，复制，粘贴，导入和导出。检视器(`inspector)` 和 `Python` 脚本调试中的创建和编辑。

`boa` - `constructor` 是用 `Python` 和 `wxPython` 库写成的。使用它之前，你必须安装了 `wxPython 2.4.0.7` 或 `wxPython` 更高的版本以及 `Python 2.1` 或 `Python` 的更高的版本。建议在使用 `boa` - `constructor` 之前，先入门 `wxPython` 。

`boa` - `constructor` 项目位于 `SourceForge` 中。 下载 `boa` - `constructor`

如何对列表排序？

在 `wxPython` 中有三个有用的方法可以对列表进行排序，在这一节，我们将按照从易到难的顺序来讨论。

在创建的时候告诉列表去排序

对一个列表控件排序的最容易的方法，是在构造函数中告诉该列表控件对项目进行排序。你可以通过使用样式标记 `wx.LC_SORT_ASCENDING` 或 `wx.LC_SORT_DESCENDING` 来实现。这两个标记导致了列表在初始显示的时候被排序，并且在 `Windows` 上，当新的项目被添加时，依然遵循所样式标记来排序。对于每个列表项的数据的排序，是基于其字符串文本的，只是简单的对字符串进行比较。如果列表是报告模式的，则排序是基于每行的最左边的列的字符串的。

基于数据而非所显示的文本来排序

有时，你想根据其它方面而非列表标签的字符串来对列表排序。在 `wxPython` 中，你可以做到这一点，但这是较为复杂的。首先，你需要为列表中的每个项目设置项目数据，这通过使用 `SetItemData(item, data)` 方法。参数 `item` 是项目在未

排序的列表中的索引，参数 `data` 必须是一个整形或长整形的值（由于C++的数据类型的限制），这就有点限制了该机制的作用。如果要获取某行的项目数据，可以使用方法 `GetItemData(item)` 。

一旦你设置了项目数据，你就可以使用方法 `SortItems(func)` 来排序项目。参数 `func` 是一个可调用的 `Python` 对象（函数），它需要两个整数。 `func` 函数对两个列表项目的数据进行比较——你不能得到行自身的引用。如果第一项比第二项大的话，函数将返回一个正整数，如果第一项比第二项小的话，返回一个负值，如果相等则返回0。尽管实现这个函数的最显而易见的方法是只对这两个项目做一个数字的比较就可以了，但是这并不唯一的排序方法。比如，数据项可能是外部字典或列表中的一个关键字，与该关键字相应的是一个更复杂的数据项，这种情况下，你可以通过比较与该关键字相应的数据项来排序。

使用 **`mixin`** 类进行列排序

关于对一个列表控件进行排序的常见的情况是，让用户能够通过在报告模式的列表的任一列上进行敲击来根据该列进行排序。你可以使用 `SortItems()` 机制来实现，但是它在保持到列的跟踪方面有点复杂。幸运的是，一个名为 `ColumnSorterMixin` 的 `wxPython` 的 `mixin` 类可以为你处理这些信息，它位于 `wx.lib.mixins.listctrl` 模块中。图13.5显示了使用该 `mixin` 类对列进行的排序。

图**13.5**

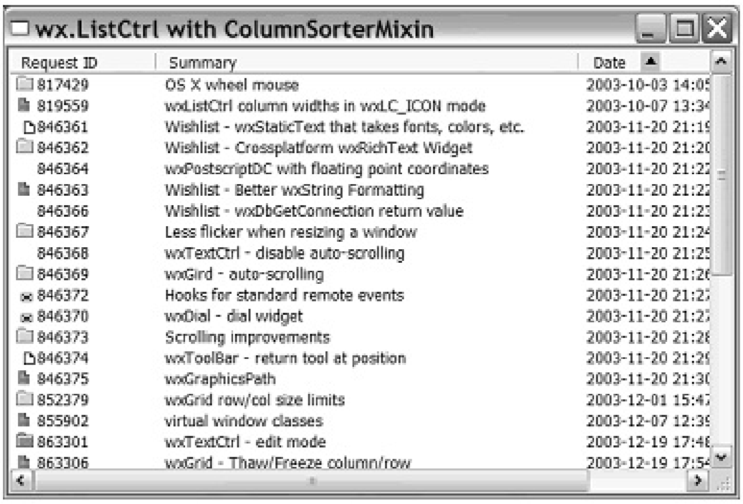


Figure 13.5
The column sorter mixin in action—notice the arrow in the date column indicating sort direction

声明这个 `mixin` 就和 `Python` 中声明任何其它的多重继承一样，如下所示：

```
import wx.lib.mixins.listctrl as listmix

class ListCtrlPanel(wx.Panel, listmix.ColumnSorterMixin):
def __init__(self, parent, log):
wx.Panel.__init__(self, parent, -1, style=wx.WANTS_CHARS)
self.list = TestListCtrl(self, tID)
self.itemDataMap = musicdata
        listmix.ColumnSorterMixin.__init__(self, 3)
```

例13.4是图13.5的例子代码

例**13.4** 使用 `mixin` 对一个报告列表进行排序

```
#!/usr/bin/python
#-*- encoding:UTF-8 -*-
import wx
import wx.lib.mixins.listctrl
import sys, glob, random
import data

class DemoFrame(wx.Frame, wx.lib.mixins.listctrl.ColumnSorterMix
in):#多重继承
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "wx.ListCtrl with ColumnSorterMixin",
                          size=(600,400))

        # load some images into an image list
        il = wx.ImageList(16,16, True)
        for name in glob.glob("smicon??.png"):
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il_max = il.Add(bmp)

        # add some arrows for the column sorter
#  添加箭头到图像列表
        self.up = il.AddWithColourMask(
            wx.Bitmap("sm_up.bmp", wx.BITMAP_TYPE_BMP), "blue")
        self.dn = il.AddWithColourMask(
            wx.Bitmap("sm_down.bmp", wx.BITMAP_TYPE_BMP), "blue"
)

        # create the list control
        self.list = wx.ListCtrl(self, -1, style=wx.LC_REPORT)

        # assign the image list to it
        self.list.AssignImageList(il, wx.IMAGE_LIST_SMALL)

        # Add some columns
        for col, text in enumerate(data.columns):
            self.list.InsertColumn(col, text)

        # add the rows
#  创建数据映射
        self.itemDataMap = {}
        for item in data.rows:
            index = self.list.InsertStringItem(sys.maxint, item[
0])
            for col, text in enumerate(item[1:]):
                self.list.SetStringItem(index, col+1, text)

            # give each item a data value, and map it back to th
```

```
e
                # item values, for the column sorter
                self.list.SetItemData(index, index)#  关联数据和映射
                self.itemDataMap[index] = item

                # give each item a random image
                img = random.randint(0, il_max)
                self.list.SetItemImage(index, img, img)

            # set the width of the columns in various ways
            self.list.SetColumnWidth(0, 120)
            self.list.SetColumnWidth(1, wx.LIST_AUTOSIZE)
            self.list.SetColumnWidth(2, wx.LIST_AUTOSIZE)
            self.list.SetColumnWidth(3, wx.LIST_AUTOSIZE_USEHEADER)

            # initialize the column sorter
            wx.lib.mixins.listctrl.ColumnSorterMixin.__init__(self,
                                                               len(da
ta.columns))

    def GetListCtrl(self):
        return self.list

    def GetSortImages(self):
        return (self.dn, self.up)

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

为了使用该 `mixin` 工作，你需要执行下面的东西：

1、扩展自 `ColumnSorterMixin` 的类（这里是 `DemoFrame` ）必须有一个名为 `GetListCtrl()` 的方法，它返回实际要被排序的列表控件。该方法被这个 `mixin` 用来得到控件的一个索引。

2、在扩展自 `ColumnSorterMixin` 的类（这里是 `DemoFrame` ）的 `__init__()` 方法中，在你调用 `ColumnSorterMixin` 的 `__init__()` 方法之前，你必须创建 `GetListCtrl()` 所要引用的列表控件。
该 `mixin` 的 `__init__()` 方法要求一个代表列表控? 械牧泻诺恼樑八怠? 3、你必须使用 `SetItemData()` 为列表中的每行设置一个唯一的数据值。

4、扩展自 `ColumnSorterMixin` 的类（这里是 `DemoFrame` ）必须有一个名为 `itemDataMap` 的属性。该属性必须是一个字典。字典中的关键性的东西是由 `SetItemData()` 设置的数据值。这些值是你想用来对每列进行排序的值的一个元组。（典型情况下，这些值将是每列中的文本）。按句话说， `itemDataMap` 本质上是将控件中的数据复制成另一种易于排序的形式。

在 `ColumnSorterMixin` 的通常用法中，你要么创建 `itemDataMap` 用来添加项目到你的列表控件，要么你首先创建 `itemDataMap` ，并用它来建造列表控件本身。

尽管配置可能有点复杂，但 `ColumnSorterMixin` 对于列的排序是一个不错的选择。

进一步了解列表控件

有时候，在你的程序中的某处你需要确定列表中的哪个项目被选择了，或者你需要通过编程来改变当前的选择以响应用户事件，或其它发生在你的程序中的一些事情。

有几个与查找列表中的一个项目的索引相关的方法，它们提供了项目的一些信息，如表13.7所示。

表13.8显示出了由 `HitTest()` 方法返回的可能的标记。实际应用中，可能返回不止一个标记。

表**13.7** 查找列表中的项目的方法

`FindItem(start, str, partial=False)`	查找第一个与 `str` 匹配的 从 `start` 的指定的索引处 符串，而非完全匹配。返回
`FindItemAtPos(start ， point, direction)`	查找与最接近位置点 `poin` 的位置。参数 `direction` `wx.LIST_FIND_DOWN, wx`
`FindItemData(start,data)`	查找项目数据（使用 `SetI` 数 `start` 同 `FindItem()`
`HitTest(point)`	返回一个(`index, flags)` 如果没有所匹配的项目，那 息。 `flags` 是一个位掩码

表**13.8** 关于 **`HitTest()`** 方法返回值中的标记

wx.LIST_HITTEST_ABOVE	位置点在列表的客户区域的上面。
wx.LIST_HITTEST_BELOW	位置点在列表的客户区域的下面。
wx.LIST_HITTEST_NOWhere	位置点在列表的客户区域中，但不属于任何项目的部分。通常这是因为它是在列表的结尾处。
wx.LIST_HITTEST_ONITEM	位置点在项目的矩形区域中，(index, flags) 中的 index 是该项目的索引。
wx.LIST_HITTEST_ONITEMICON	位置点在项目的图标区域中，(index, flags) 中的 index 是该项目的索引。
wx.LIST_HITTEST_ONITEMLABEL	位置点在项目的标签区域中，(index, flags) 中的 index 是该项目的索引。
wx.LIST_HITTEST_ONITEMRIGHT	位置点在项目右边的空白区域中。
wx.LIST_HITTEST_ONITEMSTATEICON	位置点在一个项目的状态图标中。我们这里假设列表是树形的模式，并且存在一个用户定义的状态。
wx.LIST_HITTEST_TOLEFT	位置点在列表的客户区域的左边。
wx.LIST_HITTEST_TORIGHT	位置点在列表的客户区域的右边。

至于其它的方面，还有几个方法，它们将为你提供关于所指定的项目的一些信息。方法 `GetItem()` 和 `GetItemText()` 方法我们早先已说过了，其它的见表13.9

表**13.9** 获得列表控件的项目信息的方法

`GetItemPosition(item)`
`GetItemRect(item,code= wx.LIST_RECT_BOUNDS)`
`GetNextItem(item, geometry=wx.LIST_ALL, state=wx.LIST_STATE_DONTCA`
`SetItemPosition(item, pos)`

表13.10列出了用于`GetNextItem()`的`geometry`参数的取值。`geometry`参数只用于微软`Windows`下。

表13.10 **`GetNextItem()`**的**`geometry`**参数的取值

`wx.LIST_NEXT_ABOVE`	查找显示上位于开始项目之上的下一个为指定状态的项目。
`wx.LIST_NEXT_ALL`	在列表中按索引的顺序查找下一个为指定状态的项目。
`wx.LIST_NEXT_BELOW`	查找显示上位于开始项目之下的下一个为指定状态的项目。
`wx.LIST_NEXT_LEFT`	查找显示上位于开始项目左边的下一个为指定状态的项目。
`wx.LIST_NEXT_RIGHT`	查找显示上位于开始项目右边的下一个为指定状态的项目。

表13.11列出了用于`GetNextItem()`的`state`参数的取值

表**13.11** 用于`GetNextItem()`的`state`参数的取值

`wx.LIST_STATE_CUT`	只查找所选择的用于剪贴板剪切和粘贴的项目。
`wx.LIST_STATE_DONTCARE`	查找项目，不管它当前的状态。
`wx.LIST_STATE_DropHILITED`	只查找鼠标要释放的项目。
`wx.LIST_STATE_FOCUSED`	只查找当前有焦点的项目。
`wx.LIST_STATE_SelectED`	只查找当前被选择的项目。

表13.12显示了用于改变一个项目的文本显示的方法以及用于控件项目的字体和颜色的方法。

表**13.2** 列表控件的显示属性

`GetBackgroundColour()`	处理整个列表控件的背景色。参数`col`是一个`wx.Colour`或颜色名。
`SetBackgroundColour(col)`	
`GetItemBackgroundColour(item)`	处理索引`item`所指定的项目的背景色。这个属性只用于报告模式。
`SetItemBackgroundColour(item,col)`	
`GetItemTextColour(item)`	处理索引`item`所指定的项目的文本的颜色。这个属性只用于报告模式。
`SetItemTextColour(item, col)`	
`GetTextColour()`	处理整个列表的文本的颜色。
`SetTextColour(col)`	

表13.3显示了列表控件的其它的一些方法。

表**13.3** 列表控件的其它的一些方法

`GetItemSpacing()`	返回位于图标间的空白的 `wx.Size` 。单位为像素。
`GetSelectedItemCount()`	返回列表中当前被选择的项目的数量。
`GetTopItem()`	返回可见区域顶部的项目的索引。只在报告模式中有意义。
`GetViewRect()`	返回一个 `wx.Rect` ，它是能够包含所有项目所需的最小矩形（没有滚动条）。只对图标或小图标模式有意义。
`ScrollList(dx, dy)`	使用控件滚动。参数 `dy` 是垂直量， `dx` 是水平量，单位是像素。对于图标、小图标或报告模式，单位是像素。如果是列表模式，那么单位是列数。

上面的这些表涉及了一个列表控件的大多数功能。然而到目前为止，我们所见过的所有的列表控件，它们被限制为：在程序的运行期间，它们的所有数据必须存在于内存中。在下一节，我们将讨论一个机制，这个机制仅在数据需要被显示时，才提供列表数据。

创建一个虚列表控件

让我们设想你的 `wxPython` 应用程序需要去显示包含你所有客户的一个列表。开始时你使用一个标准的列表控件，并且它工作的很好。后来人的客户列表变得越来越大，太多的客户使得你的应用程序开始出现了效率问题。这时你的程序起动所需的时间变得较长了，并占用越来越多的内存。你怎么办呢？你可以创建一个虚的列表控件。

问题的实质就是列表控件的数据处理。通常，这些数据都是从数据产生的地方将数据拷贝到列表控件中。这是潜在地浪费资源，对于一个小的列表，这好象看不出任何问题，但对于创建一个较大的列表控件，这将占用很多的内存，并导致启动变慢。

为了将一个列表控件所占的内存和启动所需的时间降到最小化， `wxPython` 允许你去声明一个虚的列表控件，这意味关于每项的信息只在控件需要去显示该项时才生成。这就防止了控件一开始就将每项存储到它的内存空间中，并且这也意味着在启动时，并没有声明完整的列表控件。同时这个方案的缺点就是虚列表中的列表项的恢复可能变得较慢。图13.6显示了一个虚列表。

例13.5显示了产生该虚列表控件的完整代码

图**13.6**

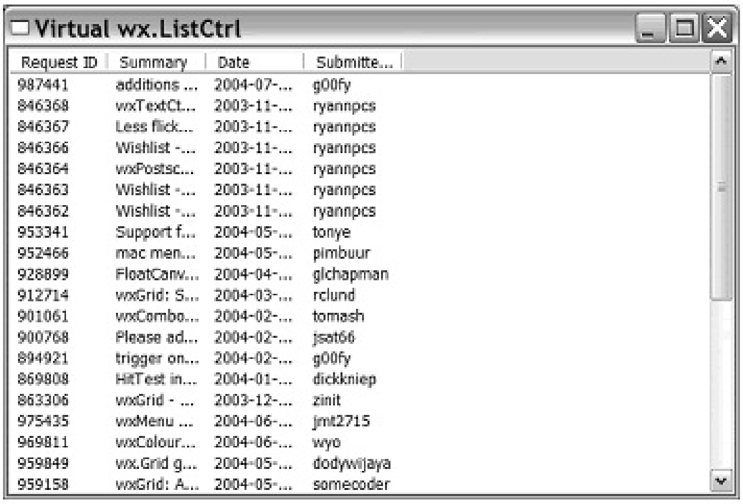


Figure 13.6
A virtual list control

例**13.5** 一个虚列表控件

```
#!/usr/bin/python
#-*- encoding:UTF-8 -*-
import wx
import sys, glob, random
import data

class DataSource:#数据源
    """
 A simple data source class that just uses our sample data items
.
 A real data source class would manage fetching items from a
 database o similar.
 """
    def GetColumnHeaders(self):
        return data.columns

    def GetCount(self):
        return len(data.rows)

    def GetItem(self, index):
        return data.rows[index]

    def UpdateCache(self, start, end):
        pass

class VirtualListCtrl(wx.ListCtrl):#1 声明虚列表
    """
 A generic virtual listctrl that fetches data from a DataSource.
 """
    def __init__(self, parent, dataSource):
        wx.ListCtrl.__init__(self, parent,
            style=wx.LC_REPORT|wx.LC_SINGLE_SEL|wx.LC_VIRTUAL)#
使用wx.LC_VIRTUAL标记创建虚列表
        self.dataSource = dataSource
        self.Bind(wx.EVT_LIST_CACHE_HINT, self.DoCacheItems)
        self.SetItemCount(dataSource.GetCount())#设置列表的大小
```

```
        columns = dataSource.GetColumnHeaders()
        for col, text in enumerate(columns):
            self.InsertColumn(col, text)

    def DoCacheItems(self, evt):
        self.dataSource.UpdateCache(
            evt.GetCacheFrom(), evt.GetCacheTo())

    def OnGetItemText(self, item, col):#得到需求时的文本
        data = self.dataSource.GetItem(item)
        return data[col]

    def OnGetItemAttr(self, item):  return None
    def OnGetItemImage(self, item): return -1

class DemoFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          "Virtual wx.ListCtrl",
                          size=(600,400))

        self.list = VirtualListCtrl(self, DataSource())

app = wx.PySimpleApp()
frame = DemoFrame()
frame.Show()
app.MainLoop()
```

这个数据源的类是一个简单的例子，它存储了我们所需要的数据项。真实情况下的数据源类还会处理从一个数据库中获取数据之类的情况，这种情况下只需要重新实现本例中的同一接口。

要创建一个虚列表，第一步就是在初始化的时候对列表控件使用 `wx.LC_VIRTUAL` 标记如#1。通常使用子类 `wx.ListCtrl` 来创建你的虚列表控件，而非仅仅使用构造函数。这是因为你需要覆盖 `wx.ListCtrl` 的一些方法，以便扩展这个虚列表。虚列表的声明类似如下：

```
class MyVirtualList(wx.ListCtrl):
def __init__(self, parent):
wx.ListCtrl.__init__(self, parent, -1,
style=wx.LC_REPORT|wx.LC_VIRTUAL)
```

有时在虚列表的初始化期间，必须调用 `SetItemCount()` 方法。这将告诉控件在数据源中存在多少数据项，这样它就可以设置适当的限制并处理滚动条。如果数据源中的数据项的数量改变了，你可以再调用 `SetItemCount()` 一次。你所覆盖的任何以 `On` 开关的方法，必须能够处理[0, `SetItemCount()` －1]间的数据。

你的虚列表控件可以覆盖其父类的三个方法，以便决定在列表控件中显示些什么。最重要的要覆盖的方法是 `OnGetItemText(item, col)` 。其中的参数 `item` 和 `col` 是要绘制的单元格的行和列，方法的返回值是显示在该单元格中的文本字符串。例如，下面的方法将只显示相关单元格的坐标。

```
def OnGetItemText(self, item, col):
return "Item %d, column %d" % (item, col)
```

如果你想在一行中显示一个图像，你需要覆盖 `OnGetItemImage(item)` 。它的返回值是较早声明的列表控件的图像列中的一个整数索引。如果你没有覆盖这个方法，那么基类版本的 `OnGetItemImage` 将返回－1，这表明不显示图像。如果你想改变行的一些显示属性，那么你可以覆盖 `OnGetItemAttr(item)` 方法， `item` 是行的索引，该方法返回类 `wx.ListItemAttr` 的一个实例。该类有一些 `get` 和 *`set`* 方法可以用来设置行的颜色、对齐方式等等显示属性。

如果你的虚列表所基于的数据改变了，而你想更新显示，那么你可以使用该列表控件的 `RefreshItem(item)` 来重绘特定的行。相关的方法 `RefreshItems(itemFrom,itemTo)` 重绘位于索引 `itemFrom` 和 `itemTo` 间的所有行。

为了对数据源中的数据的获取提供优化帮助，对于要显示一页新的数据，虚列表控件会发送 `EVT_LIST_CACHE_HINT` 事件。这将给你的数据源一个时机用以从数据库（或另处）一次获取几个记录并保存它们。这样就使得随后的 `OnGetItemText()` 执行的更快。

本章小结

1、列表控件是 `wxPython` 用于显示列表信息的窗口部件。它比简单的列表框部件要更复杂且有完整的特性。列表控件是类 `wx.ListCtrl` 的实例。列表控件可以显示为图标模式，每个图标下都有一个项目文本，也可以显示为带有小图标的小图标模式等等。在列表模式中，元素按列显示，在报告模式中，以多列的格式显示列表，每列都有列标签。

2、用于列表控件的图像是由一个图像列表管理的，图像列表是一个可经由索引来访问的一个图像的数组。列表控件可以为不同的列表模式维护各自的图像列表，这使得能够容易地在模式间切换。

3、你可以使用 `InsertStringItem(index,label)` 方法来插入文本到列表中，使用 `InsertImageItem(index, imageIndex)` 方法插入图像到列表中。要一次做上面两件事，可以使用 `InsertImageStringItem(index,label,`

- `imageIndex)` 。要对报告模式的列表添加列，可以使用 `InsertColumn(col, heading, format=` " `wx.LIST_FORMAT_LEFT, widt` -1)方法。一旦已经添加了列后，你就可以使用 `SetStringItem(index, col, label, imageId=` -1)方法为新的列增加文本。

4、列表控件产生的几个事件可以被绑到程序的动作。这些事件项属于类`wx.ListEvent`。通常的事件类型包括`EVT_LIST_Insert_ITEM, EVT_LIST_ITEM_ACTIVATED,`和`EVT_LIST_ ITEM_S`。

5、如果列表控件声明时使用了`wx.LC_EDIT_LABELS`标记，那么用户就可以编辑列表项的文本。编辑的确认是通过按下回车键或在列表中敲击完成的，也可以通过按下`Esc`键来取消编辑。

6、你可以通过在声明列表时使用`wx.LC_SORT_ASCENDING`或`wx.LC_SORT_DESCENDING`来排序列表。这将按照项目的字符串的顺序来排序列表。在报告模式中，这将根据0列的字符串来排序。你也可以使用`SortItems(func)`方法来创建你自定义的排序方法。对于报告模式的列表，`mixin`类`wx.lib.mixins.listctrl.ColumnSorterMixin`给了你根据用户所选择的列来排序的能力。

7、使用了标记`wx.LC_VIRTUAL`声明的列表控件是一个虚列表控件。这意味着它的数据是当列表中的项目被显示时动态地确定的。对于虚列表控件，你必须覆盖`OnGetItemText(item, col)`方法以返回适当的文本给所显示的行和列。你也可以使用`OnGetItemImage(item)`和`OnGetItemAttr(item)`方法来返回关于每行的图像或列表的显示属性。如果数据源的数据改变了，你可以使用`RefreshItem(item)`方法来更新列表的某个行或使用`RefreshItems(itemFrom, itemTo)`方法来更新多个行。

最终，你的数据将变得复杂得不能放在一个简单的列表中。你将会需要类似二维的电子表格样式的东西，这就是网格控件，我们将在下一章进行讨论。

第十四章 网格控件

本章内容包括:

- 创建网格(`grid)`
- 添加行和单元格(`cell),` 并且处理列的首部
- 使用一个自定义的单元格(`cell)` 描绘器(`renderer)`
- 创建自定义的编辑器
- 捕获用户事件

网格控件大概是 `wxPython` 中最复杂和最灵活的一个窗口部件。在这一章，你将有机会接触到这个控件的许多特性。我们将讨论如何输入数据到网格控件以及如何处理该控件的显示属性，并且我们还将讨论自定义编辑器和描绘器。网格控件使你能够在一个类似电子表格格式的网格中显示表格数据。该控件允许你为行和列指定标签，以及通过拖动网格线来改变网格的大小，并且可以为每个单元格单独指定字体和颜色属性。

在最常见的情况下，你一般会显示一个简单的字符串值。然而，你也能为任一单元格指定一个自定义的描绘器，以使你能够显示不同的数据；你可以有编辑表中的单元格，并且对不同的数据使用不同类型的编辑器。你还能够创建你自己自定义的描绘器和编辑器，这使得你在单元格数据的显示和处理上可以非常的灵活，几乎没有什么限制。网格控件还有大量的鼠标和键盘事件，你可以程序中捕获它们并用来触发相关的代码。

我们将通过展示两个创建 `wxPython` 网格的方法来作为我们讨论的开始。

创建你的网格

网格控件是用以显示一个二维的数据集的。要使用该控件显示有用的信息，你需要告诉该控件它工作所基于的是什么数据。在 `wxPython` 中，有两种不同的机制用于在网格控件中处理数据，它们之间在处理数据的添加，删除和编辑的方式上有些许的不同。

- 网格控件可以直接处理每行和每列中的值。
- 数据可以通过使用一个网格表(`grid table)` 来间接地处理。

较简单的一种是使用网格控件直接处理值。在这种情况下，网格维护着数据的一份拷贝。在这种情况下，如果有大量的数据或你的应用程序已经有了一个现存的网格类的数据结构，那么这可能显得比较笨拙。如果是这样，你可以使用一个网格表来处理该网格的数据。参见第5章来回顾一下在 `MVC` 架构中，网格表是如何被作为一个模型的。

如何创建一个简单的网格？

尽管网格控件有大量的方法用于控件精确的显示和数据的管理，但时开始使用一个网格控件是十分简单的。图14.1显示了一个简单的网格，其中的单元格中添加了一些字符串数据。

图**14.1**

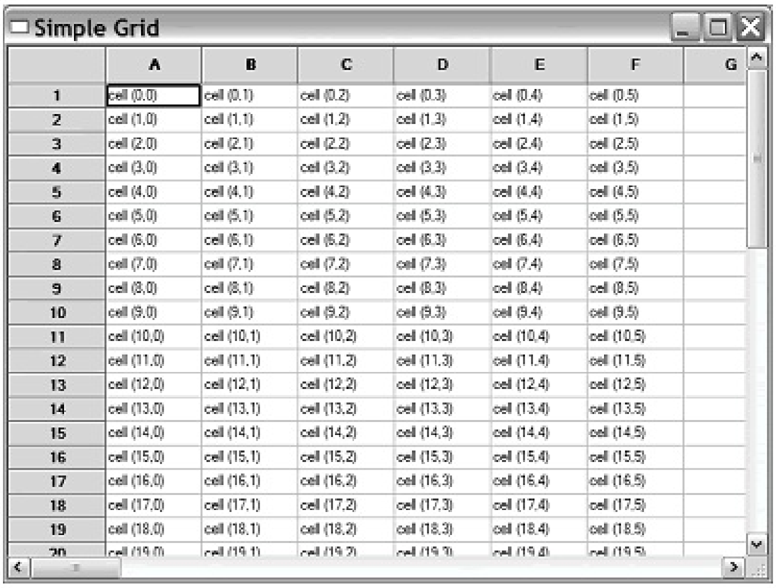


Figure 14.1
Abasic grid control

网格控件是类 `wx.grid.Grid` 的实例。由于网格类及相关类的尺寸的原因，实际中许多的程序都不使用它， `wxPython` 的网格类存在于它们自己的模块中，它们不会被自动导入到核心的名字空间中。 `wx.grid.Grid` 的构造函数类似于其它的控件的构造函数。

```
wx.grid.Grid(parent, id, pos=wx.DefaultPosition,
size=wx.DefaultSize, style=wx.WANTS_CHARS,
        name=wxPanelNameStr)
```

其中的所有的参数与 `wx.Window` 的构造函数是相同的，并且有相同的意义。 `wx.WANTS_CHARS` 样式是网格的默认样式，除此之外， `wx.grid.Grid` 没有为自己定义特别的样式。由于网格类的复杂性，所以在程序中，你一般要自定义网格类的一个子类来实现一个网格，而非直接使用 `wx.grid.Grid` 的一个实例。

和我们所见过的别的控件不同，调用该构造函数不足以创建一个可用的网格。有两个方法用以初始化网格

- `CreateGrid()` * `SetTable()`

在这一节，我们将讨论一个方法，第二个方法将在网格表的讨论中提及。

要显式地初始化网格，可以使用方法 `CreateGrid(numRows, numCols, selmode=wx.grid.Grid.SelectCells)` 。这个方法应该在构造函数之后被直接地调用，并用必须在网格被显示之前调用。参

数 `numRows, numCols` 指定了网格的初始大小。参数 `selmode` 指定了网格中单元格的选择模式，默认值是 `wx.grid.Grid.SelectCells` ，意思是一次只选择一个单元格。其它的值有 `wx.grid.Grid.SelectRows` ，意思是一次选择整个行， `wx.grid.Grid.SelectionColumns` ，意思是一次选择整个列。创建之后，你可以使用方法 `GetSelectionMode()` 来访问选择模式，并且你可以使用方法 `SetSelectionMode(mode)` 来重置模式。你还可以使用方法 `GetNumberCols()` 和 `GetNumberRows()` 来得到行和列数。

在内部，使用 `CreateGrid()` 初始化网格之后， `wxPython` 设置了一个二维的字符串数组。一旦网格被初始化了，你就可以使用方法 `SetCellValue(row, col, s)` 来放置数据。其中参数 `row, col` 是要设置的单元格的坐标，s是要在该坐标处显示的字符串文本。如果你想获取特定坐标处的值，你可以使用函数 `GetCellValue(row, col)` ，该函数返回字符串。要一次清空整个网格，你可以使用方法 `ClearGrid()` 。例14.1显示了产生图14.1的代码。

例**14.1** 使用 **`ClearGrid()`** 创建的一个示例网格

```
import wx
import wx.grid

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Simple Grid",
                          size=(640,480))
        grid = wx.grid.Grid(self)
        grid.CreateGrid(50,50)
        for row in range(20):
            for col in range(6):
                grid.SetCellValue(row, col,
                                  "cell (%d,%d)" % (row, col))

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

`CreateGrid()` 和 `SetCellValue()` 仅限于你的网格数据是由简单字符串组成的情况。如果你的数据更加的复杂或表特别大的话，更好的方法是创建一个网格表，这将随后讨论。

如何使用网格表来创建一个网格？

对于较复杂的情况，你可以将你的数据保存在一个网格表中，网格表是一个单独的类，它存储数据并与网格控件交互以显示数据。推荐在下列情况下使用网格表：

- 网格的数据比较复杂 数据存储在你的系统中的另外的对象中 网格太大以致于不能一次整个被存储到内存中

在第5章中，我们在 `MVC` 设计模式中讨论了网格表以及在你的应用程序中实现一个网格表的不同方法。在本章，我们将重点放在对网格表的使用上。图14.2显示了使用网格表创建的一个网格。

图**14.2**

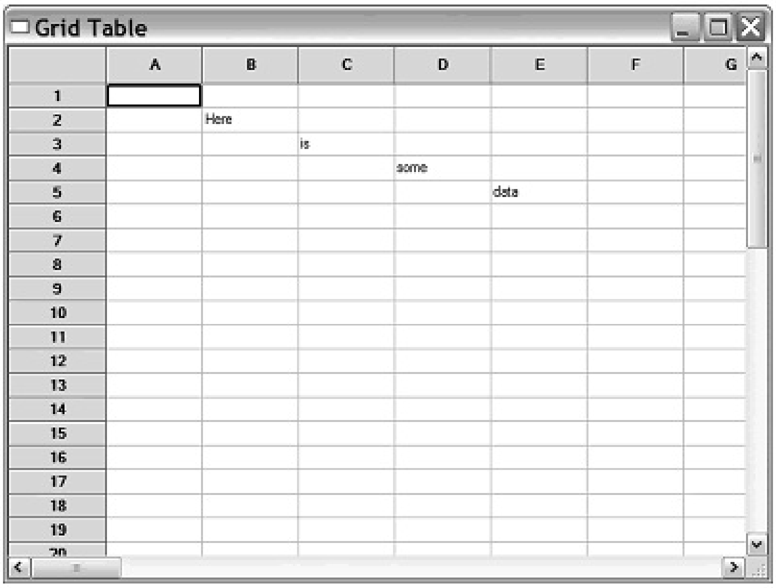


Figure 14.2
A simple grid using the grid table mechanism

要使用一个网格表，你需要要创建 `wx.grid.PyGridTableBase` 的子类。该子类必须覆盖父类 `wx.grid.GridTableBase` 的一些方法。例14.2显示了用于创建图14.2的代码。

例**14.2** 关于使用网格表机制的代码

```
#-*- encoding:UTF-8 -*-
import wx
import wx.grid

class TestTable(wx.grid.PyGridTableBase):#定义网格表
    def __init__(self):
        wx.grid.PyGridTableBase.__init__(self)
        self.data = { (1,1) : "Here",
                      (2,2) : "is",
                      (3,3) : "some",
                      (4,4) : "data",
                      }

        self.odd=wx.grid.GridCellAttr()
        self.odd.SetBackgroundColour("sky blue")
        self.odd.SetFont(wx.Font(10, wx.SWISS, wx.NORMAL, wx.BOL
D))

        self.even=wx.grid.GridCellAttr()
        self.even.SetBackgroundColour("sea green")
        self.even.SetFont(wx.Font(10, wx.SWISS, wx.NORMAL, wx.BO
LD))
```

```
    # these five are the required methods
    def GetNumberRows(self):
        return 50

    def GetNumberCols(self):
        return 50

    def IsEmptyCell(self, row, col):
        return self.data.get((row, col)) is not None

    def GetValue(self, row, col):#为网格提供数据
        value = self.data.get((row, col))
        if value is not None:
            return value
        else:
            return ''

    def SetValue(self, row, col, value):#给表赋值
        self.data[(row,col)] = value

    # the table can also provide the attribute for each cell
    def GetAttr(self, row, col, kind):
        attr = [self.even, self.odd][row % 2]
        attr.IncRef()
        return attr

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Table",
                          size=(640,480))

        grid = wx.grid.Grid(self)

        table = TestTable()
        grid.SetTable(table, True)

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

在例14.2中，所有特定于应用程序的逻辑都已被移到了网格表类，所以这里就没有必须创建一个自定义的 `wx.grid.Grid` 的子类。

要使网格表有效，你必须覆盖5个方法。表14.1列出了这些方法。在这一章中，我们还会看到其它你能覆盖的方法，你可以覆盖它们以给于你的表更多的功能。

表14.1 `wx.grid.GridTableBase` 中需要被覆盖的方法

GetNumberCols()	返回显示在网格中的列的数目
GetNumberRows()	返回显示在网格中的行的数目
GetValue(row, col)	返回坐标(row, col) 处的值
IsEmptyCell(row, col)	如果坐标(row, col) 处的单元格为空的话，返回 True 。否则返回 False 。

SetValue(row, col, value) ：如果你需要的话，它使你能够更新你底层的数据结构以匹配用户的编辑。对于一个只读的表，你仍然需要声明该方法，但是你可以通过 pass 来使它什么也不做。该方法在当用户编辑一个单元格时自动被调用。

要将网格表实例附着于你的表的实例，要调用 SetTable(table,takeOwnership=False,selmode=wx.grid.Grid.SelectCells) 方法。其中参数 table 是你的 wx.grid.PyGridTableBase 的实例。参数 takeOwnership 使得网格控件拥有这个表。如果 takeOwnership 为 True ，那么当网格被删除时，该表也被 wxPython 系统删除。参数 selmode 作用等同于在 CreateGrid() 中的作用。

还有一些其它的方法你可以覆盖，以处理网格的各部分，而非表的数据。在本章的稍后部分，我们将讨论这些方法中的一些。并且，我们将看到在某些情况中，使用 SetTable 创建的表的行为与使用 CreateGrid() 创建的表的行为是不同的。

你能够覆盖的另一个方法是 Clear() ，它在当对网格调用 ClearGrid() 时被调用，如果适当的话，你可以覆盖该方法来清除潜在的数据源。在网格中置入数据了以后，你现在可以开始对网格作各种有兴趣的事情了。在下一节，我们将给你展示如何处理网格的外观。

使用网格工作

一旦网格被创建并初始化了，你就可以用很多不同的方法来处理它了。单元格、行或列可以被添加和删除。你可以增加首部，改变一行或一列的大小，并可以用代码的方式来改变网格的可见部分或被选择的部分。下面的几节，我们将涉及这些内容。

如何添加、删除行，列和单元格？

在网格被创建之后，你仍然可以添加新的行和列。注意，依据网格的创建方法不同，该机制的工作也不同。你可以使用 AppendCols(numCols=1) 方法在你的网格的右边增加一列。使用 AppendRows(numRows=1) 在网格的底部增加一行。

如果不是想在网格的行或列的最后添加一行或一列，你可以使用方法 InsertCols(pos=0, numCols=1) 或 InsertRows(pos=1, numRows=1) 来在指定位置添加。其中参数 pos 代表被添加的新元素中第一个的索引。如果参数 numRows 或 numCols 大于1 ，那么有更多的元素被添加到起始位置的右边（对于列来说），或起始位置的下边（对于行来说）。

要删除一行或一列，你可以使用方法 `DeleteCols(pos=0, numCols=1)` 和 `DeleteRows(pos=0, numRows=1)` 。其中参数 `pos` 是要被删除的行或列的第一个的索引。

如果网格是使用 `CreateGrid()` 方法被初始化的，那么上面讨论的方法总是可以工作的，并且在新的行或列中创建的单元格是以一个空字符串从为初始值的。如果网是使用 `SetTable()` 方法被初始化的，那么网格表必须支持对表的改变。

要支持改变，你的网格表要对同样的改变方法进行覆盖。例如，如果你对你的网格调用了 `InsertCols()` 方法，那么网格表也必须声明一个 `InsertCols(pos=0, numCols=1)` 方法。该网格表的这个方法返回布尔值 `True` 表示支持改变，返回 `False` 则否决改变。例如，要创建一个只允许被扩展到50行的一个表，可以在你的网格表中写上下面的方法。

```
def AppendRows(self, numRows=1):
    return (self.GetRowCount() + numRows)  = 50
```

某些对网格的改变不会立即被显示出来，而是要等待网格被刷新。你可能通过使用 `ForceRefresh()` 方法来触发一个即时的刷新。在通常情况下，如果你用代码的方式来改变你的网格，则改变不会立即显示出来，那么插入对 `ForceRefresh()` 方法的调用可以确保你的改变即时的显示出来。

如果你在对一个网格作一个大量的改变，而你在改变期间不想让网格的显示产生闪烁的话，你可以通过使用 `BeginBatch()` 方法来告诉该网格去作一个批量的处理。该方法将针对网格作一个内在的增量计数。你也必须在批量的任务之后调用 `EndBatch()` ——该方法针对网格作一个内在的减量计数。当计数值比0大时，表明正处于开始和结束计数之间，网格这时不会重绘。如果必要的话，你还可以在批量处理中再嵌套批量处理。同样，在全部的批量处理没有完成时，网格不会重绘。

如何处理一个网格的行和列的首部？

在一个 `wxPython` 的网格控件中，每行和每列都有它们自己的标签。默认情况下，行的标签是数字，从1开始。列的标签是字母，从A开始。 `wxPython` 提供了一些方法来改变这些标签。图14.3显示了一个带有首部标签的网格。

图**14.3**

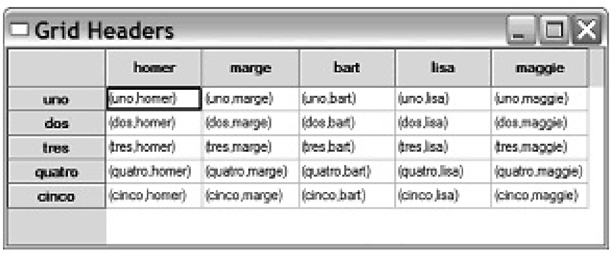


Figure 14.3
A sample grid with
custom labels created

例子14.3显示了产生图14.3的代码。其中网格是用 `CreateGrid()` 初始化的。

例**14.3** 带自定义标签的一个非模式的网格

```
import wx
import wx.grid

class TestFrame(wx.Frame):

    rowLabels = ["uno", "dos", "tres", "quatro", "cinco"]
    colLabels = ["homer", "marge", "bart", "lisa", "maggie"]

    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Headers",
                          size=(500,200))
        grid = wx.grid.Grid(self)
        grid.CreateGrid(5,5)
        for row in range(5):
            #1 start
            grid.SetRowLabelValue(row, self.rowLabels[row])
            grid.SetColLabelValue(row, self.colLabels[row])
            #1 end
            for col in range(5):
grid.SetCellValue(row, col,
                        "(%s,%s)" % (self.rowLabels[row], self.c
olLabels[col]))

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

正如添加和删除行一样，改变标签也是根据网格的类型而不同的。对于使用 `CreateGrid()` 创建的网格，要使用 `SetColLabelValue(col, value)` 和 `SetRowLabelValue(row, value)` 方法来设置标签值，如#1所示。参数 `col` 和 `row` 是列和行的索引， `value` 是要显示在标签中的字符串。要得到一行或一列的标签，使用 `GetColLabelValue(col)` 和 `GetRowLabelValue(row)` 方法。

对于使用外部网格表的一个网格控件，你可以通过覆盖网格表的 `GetColLabelValue(col)` 和 `GetRowLabelValue(row)` 方法来达到相同的作用。为了消除混淆，网格控件在当它需要显示标签并且网格有一个关联的表时，内在地调用这些方法。由于返回值是动态地由你在覆盖的方法中所写的代码决定的，所以这里不需要覆盖或调用 `set` 方法。不过 *`set`* 方法仍然存在——`SetColLabelValue(col, value)` 和 `SetRowLabelValue(row, value)` ——但是你很少会使用到，除非你想让用户能够改变潜在的数据。通常，你不需要 `set` *方法。例14.4显示了如何改变网格表中的标签——这个例子产生与上一例相同的输出。

例**14.4** 带有自定义标签的使用了网格表的网格

```
import wx
import wx.grid

class TestTable(wx.grid.PyGridTableBase):
    def __init__(self):
        wx.grid.PyGridTableBase.__init__(self)
        self.rowLabels = ["uno", "dos", "tres", "quatro", "cinco
"]
self.colLabels = ["homer", "marge", "bart", "lisa", "maggie"]

    def GetNumberRows(self):
        return 5

    def GetNumberCols(self):
        return 5

    def IsEmptyCell(self, row, col):
        return False

    def GetValue(self, row, col):
        return "(%s,%s)" % (self.rowLabels[row], self.colLabels[
col])

    def SetValue(self, row, col, value):
        pass

    def GetColLabelValue(self, col):#列标签
        return self.colLabels[col]

    def GetRowLabelValue(self, row):#行标签
        return self.rowLabels[row]

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Table",
                          size=(500,200))
        grid = wx.grid.Grid(self)
        table = TestTable()
        grid.SetTable(table, True)

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

默认情况下，标签是居中显示的。但是你也可以使用 `SetColumnLabelAlignment(horiz, vert)` 和 `SetRowLabelAlignment(horiz` 来改变这个行为。其中参数 `horiz` 用以控制水平对齐方式，取值

有`wx.ALIGN_LEFT, wx.ALIGN_CENTRE`或`wx.ALIGN_RIGHT`。参数`vert`用以控制垂直对齐方式，取值有`wx.ALIGN_TOP, wx.ALIGN_CENTRE,`或`wx.ALIGN_BOTTOM`。

行和列的标签区域共享一套颜色和字体属性。你可以使用`SetLabelBackgroundColour(colour)`，`SetLabelFont(font), and SetLabelTextColour(colour)`方法来处理这些属性。参数`colour`是`wx.Colour`的一个实例或`wxPython`会转换为颜色的东西，如颜色的字符串名。参数`font`是`wx.Font`的一个实例。与`set`相应的*`get`*方法有`GetLabelBackgoundColour(), GetLabelFont()`，和`GetLabelTextFont()`。

如何管理网格元素的尺寸？

网格控件提供了几个不同的方法来管理单元格、行和列的尺寸。在这一节，我们将讨论这些方法。图14.4显示了一些用来改变一个特定的单元格的尺寸的方法。

例14.5显示了创建了一个带有可调节大小的单元格、行和列的网格。

图**14.4**

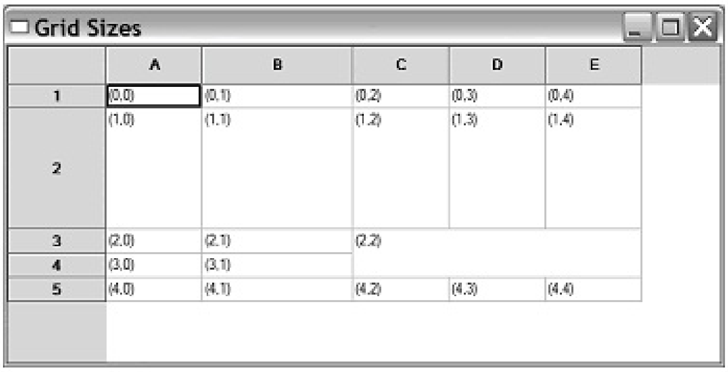


Figure 14.4
A sample grid, showing resized cells, rows, and columns

例**14.5** 可调整尺寸的单元格的示例代码

```
import wx
import wx.grid

class TestFrame(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Sizes",
                                 size=(600,300))
        grid = wx.grid.Grid(self)
        grid.CreateGrid(5,5)
        for row in range(5):
            for col in range(5):
                grid.SetCellValue(row, col, "(%s,%s)" % (row, co
l))

grid.SetCellSize(2, 2, 2, 3)
grid.SetColSize(1, 125)
        grid.SetRowSize(1, 100)

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

改变单元格的尺寸

一个作用于单元格尺寸的基本的方法是使它跨多行或多列，类似于 `HTML` 的 `rowspan` 和 `colspan` 。要达到这种效果，在 `wxPython` 中可以使用方法 `SetCellSize(row, col, num_rows, num_cols)` 。该方法设置坐标 `row,col` 处的单元格跨 `num_rows` 行和 `num_cols` 列。在通常的情形下，每个单元格占据一行和一列，要使用单元格不止占据一行或一列，你需要给参数 `num_rows, num_cols` 大于1的值。参数 `num_rows, num_cols` 的值小于等于0会导致错误。如果你的设置使得一个单元格的尺寸与另一个早先声明为跨越的单元格的尺寸相重叠时，早先的这个单元格的尺寸会重置为占据一行和一列。你也能够使用方法 `SetCellOverflow(row, col, allow)` 方法来关闭单元格的跨越显示。只要在该方法中使用 `pass` 就可以阻止单元格跨越了，即使已经使用了 `SetCellSize()` 方法来设置它的尺寸。

调整网格的尺寸的一个更加典型的方法是基于一行或一列来处理其像素尺寸。你可以使用 `SetColSize(col, width)` 和 `SetRowSize(row, height)` 方法来改变一列或一行的宽度。当然，你可以使用 `GetColSize(col)` 或 `GetRowSize(row)` 来确定一列或一行的当前尺寸。

设置默认尺寸

你可以通过改变所有的行和列的默认尺寸来改变整个网格的尺寸。方法如下：

```
SetDefaultColSize(width, resizeExistingCols=False)
SetDefaultRowSize(height, resizeExistingRows=False)
```

其中的第一个参数是以像素为单位的新的默认尺寸。如果第二个布尔参数的值是 `True` ，那么当前存在的所有行或列立即被调整到新的默认尺寸。如果第二个参数的值为 `False` ，那么这个新的默认尺寸仅被应用于新添加的行或列。通常，设置新的默认值是在初始化的开头，甚至是在调用 `CreateGrid()` 或 `SetTable()` 之前。你可以使用 `GetDefaultColSize()` 和 `GetDefaultRowSize()` 方法来得到当前的默认尺寸。

设置默认尺寸与为单个行或列设置尺寸相比，有一个性能上的问题。对于存储默认值， `wxPython` 只需要存储这两个整数。如果你将单个行或列设置到一个非默认的尺寸， `wxPython` 切换并将每个行或列的尺寸存储到一个数组中。如果你的表是非常的大的话，这将明显地占用很多的内存，因此这是需要注意的。

有时，你想为一行或一列设置一个最小的尺寸，以便不用担心程序的某个方法的调用或用户对网格线的拖动会致使该行或列变得更小。

在 `wxPython` 中，你可以对一个网格的宽度设置最小值或为单独的行和列分别设置最小尺寸值。要改变整个网格的最小尺寸，可以使用方法 `SetColMinimalAcceptableWidth(width)` 或 `SetRowMinimalAcceptableHeig` 。其中的参数是针对所有行或列的最小的像素尺寸。要一行一行的设置最小尺寸，使用方法 `SetColMinimalWidth(col, width)` 或 `SetRowMinimalHeight(row, height)` 。其中第一个参数是要调整尺寸的项目的索引，第二个参数是以像素为单位的新的尺寸。单个的行的最小尺寸必须比最小的网格尺寸大，如果单个的行的最小尺寸被设置了的话。上面的 `set` 方法都有一个相应的 *`get`* 方法：

- `GetColMinimalAcceptableWidth()` *`GetRowMinimalAcceptableHeight()`* `GetColMinimalWidth(col)` * `GetRowMinimalHeight(row)`

设置标签的尺寸

网格上的标签区域有一套单独的调整尺寸的函数。在这种情况下，你是在设置行标签的宽度和列标签的高度，意思就是，把列标签作为一个特殊的行，把行标签作为一个特殊的列。 `set` *方法有 `SetRowLabelSize(width)` ，它设置行标签的宽度， `SetColLabelSize(height)` ，它设置列标签的高度。你可以使用相应的 `GetRowLabelSize()` 和 `GetColLabelSize()` 方法来得到这些尺寸。

通常，你不会关心单元格的实际的像素尺寸，你希望它们被自动调整到足够显示你的数据的大小。在 `wxPython` 中，你可以通过使用 `AutoSize()` 方法来自动调整整个网格的尺寸。该方法使得所有的行和列的尺寸与它们中的内容相适应。你也可以对单个的行或列使用 `AutoSizeColumn(col, setAsMin=True)` 和 `AutoSizeRow(row, setAsMin=Tr` 来使它们的尺寸自动与其中的内容相适应。如果参数 `setAsMin` 为 `True` ，那么新的自动的尺寸将作为该行或列的最小尺寸。 `AutoSizeColumns(setAsMin=True)` 和 `AutoSizeRows(setAsMin=True)` 自动调整所有的列和行的尺寸。

你也可以让用户通过拖动标签单元格的边框来调整行的尺寸。用于实现这种行为的主要的方法如下：

- `EnableDragColSize(enable=True)`：控制用户能否通过拖动边框来改变标签的宽度
- `EnableDragRowSize(enable=True)`：控制用户能否通过拖动边框来改变标签的高度
- `EnableDragGridSize(enable=True)`：控制用户能否通过拖动边框一次性改变标签的宽度和高度

下面的方法是上面方法的相应的使拖动无效的简便的方法：

- `DisableDragColSize()`
- `DisableDragRowSize()`
- `DisableDragGridSize()`

下面的一套方法用以判断能否拖动：

- `CanDragColSize()`
- `CanDragRowSize()`
- `CanDragGridSize()`

如何管理哪些单元格处于选择或可见状态？

在网格控件中，用户可以选择一个或多个单元格。在 `wxPython` 中，有几个方法让你能够处理多选的情况。

在下面的几个情况中，网格控件中的被选择的项可以是0个或多个：

- 单个的处于选择状态的单元格
- 被选择的行
- 被选择的行
- 被选择的由单元格组成的块

用户可以通过命令或在单元格、行或列标签上的敲击，或拖动鼠标来选择多组单元格。要确定网格中是否有被选择的单元格，可能使用方法 `IsSelection()` ，如果有则该方法返回 `True` 。你可以通过使用 `IsInSelection(row, col)` 方法来查询任意一个特定的单元格当前是否处于选择状态中，如果是则返回 `True` 。

表14.2显示了几个方法，它们得到当前被选择的内容并返回给你。

表**14.2** 返回当前被选择的单元格的集的方法

GetSelectedCells()	返回包含一些单个的处于选择状态的单元格的一个 Python 列表。在这个列表中的每个项目都是一个 (row, col) 元组。
GetSelectedCols()	返回由通过敲击列的标签而被选择的列的索引组成的一个 Python 列表。
GetSelectedRows()	返回由通过敲击行的标签而被选择的列的索引组成的一个 Python 列表。
GetSelectionBlockTopLeft()	返回包含一些被选择的由单元格组成的块的一个 Python 列表。其中的每个元素都时一个(row, col) 元组，(row, col) 元组是每块的左上角。
GetSelectionBlockBottomRight()	返回包含一些被选择的由单元格组成的块的一个 Python 列表。其中的每个元素都时一个(row, col) 元组，(row, col) 元组是每块的右下角。

这儿也有几个用于设置或修改选择状态的方法。第一个是 ClearSelection() ，它清除当有的被选状态。在该方法被调用以后， IsSelection() 返回 False 。你也可以做一个相反的动作，就是使用 SelectAll() 选择所有的单元格。你也可以使用方法 SelectCol(col, addToSelected=False) 和 SelectRow(row, addToSelecte 来选择整列或整行。在这两个方法中，第一个参数是要选择的行或列的索引。如果参数 addToSelected 为 True ，所有另外被选择的单元格仍然处于被选状态，并且该行或列也被增加到已有的选择中。如果参数 addToSelected 为 False ，那么所有另外被选择的单元格解除被选状态，而新的行或列代替它们作为被选择对象。同样地，你也可以使用方法 SelectBlock(topRow, leftCol, bottomRow, rightCol, addToSelected=Fa 来增加一个对一块范围的选择，前面四个参数是所选的范围的对角， addToSelected 参数的作用同前一样。

你也可以使用 IsVisible(row, col, wholeCellVisible=True) 方法来得到一个特定的单元格在当前的显示中是否是可见的。如果该单元格当前显示在屏幕上了（相对于处在一个可滚动的容器的不可见部分而言），那么该方法返回 True 。如果参数 wholeCellVisible 为 True ，那么单元格要整个都是可见的，方法才返回 True ，如果参数 wholeCellVisible 为 False ，则单元格部分可见，方法就会返回 True 。方法 MakeCellVisible(row, col) 通过滚动确保了指定位置的单元格是可见的。

除了被选的单元格外，网格控件也有一个光标单元格，它代表获得当前用户焦点的单元格。你可以使用 GetGridCursorCol() 和 GetGridCursorRow() 方法来确定光标的当前位置，这两个方法返回整数的索引值。你可以使用 SetGridCursor(row, col) 方法来显式地放置一个光标。该方法除了移到光标外，它还隐式地对新的光标位置调用了 MakeCellVisible 。

表14.3说明了在网格坐标和显示器坐标之间作转换的网格控件的方法。

表**14.3** 坐标转换方法

`BlockToDeviceRect(topLeft, bottomRight)`	参数 `topLeft, bottomRight` 单元格的坐标（(`row, col)` 元组的形式）。返回值是一个 `wx.Rect` ， `wx.Rect` 使给定的网格坐标所包围的矩 的设备像素坐标。
`CellToRect(row, col)`	返回一个 `wx.Rect` ， `wx.Rect` 的标是相对网格坐标(`row, col)` 处的单元格的器的坐标。
`XToCol(x)`	返回包含x坐标（该坐标是相于容器的）的列的索引。如没有这样的列，则返回 `wx.NOT_FOUND` 。
`XToEdgeOfCol(x)`	返回右边缘最接近给定的x坐的列的整数索引。如果没有样的列，则返回 `wx.NOT_FOUND` 。
`YToRow(y)`	返回包含y坐标（该坐标是相于容器的）的行的索引。如没有这样的行，则返回 `wx.NOT_FOUND` 。
`YToEdgeOfRow(y)`	返回底边缘最接近给定的y坐的行的整数索引。如果没有样的行，则返回 `wx.NOT_FOUND` 。

你可以使用上面这些方法来对网格单元格上的鼠标敲击的位置作转换。

如何改变一个网格的单元格的颜色和字体?

正如其它的控件一样，这儿也有一些属性方法，你可以用来改变每个单元格的显示属性。图14.5是个示例图片。例14.6显示了产生图14.5的代码。注意其中的针对特定单元格的网格方法和 `wx.grid.GridCellAttr` 对象的创建方法的用法。

图**14.5**

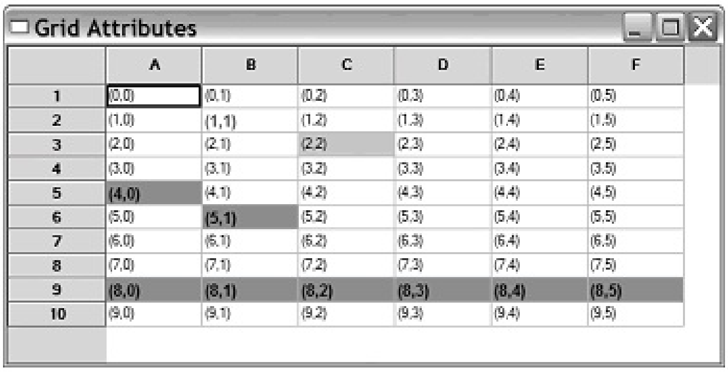


Figure 14.5
A sample usage of the grid attribute methods

例**14.6** 改变网格的单元格的颜色

```
import wx
import wx.grid

class TestFrame(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Attributes",
                          size=(600,300))
        grid = wx.grid.Grid(self)
        grid.CreateGrid(10,6)
        for row in range(10):
            for col in range(6):
                grid.SetCellValue(row, col, "(%s,%s)" % (row, co
l))

        grid.SetCellTextColour(1, 1, "red")
        grid.SetCellFont(1,1, wx.Font(10, wx.SWISS, wx.NORMAL, w
x.BOLD))
        grid.SetCellBackgroundColour(2, 2, "light blue")

        attr = wx.grid.GridCellAttr()
        attr.SetTextColour("navyblue")
        attr.SetBackgroundColour("pink")
        attr.SetFont(wx.Font(10, wx.SWISS, wx.NORMAL, wx.BOLD))

        grid.SetAttr(4, 0, attr)
        grid.SetAttr(5, 1, attr)
        grid.SetRowAttr(8, attr)

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

我们将通过讨论用于设置整个网格默认值的方法作为开始。你可以使用 `SetDefaultCellAlignment(horiz, vert)` 方法来为网格中所有的单元格设置默认的对齐方式，其中 `horiz` 的取值

有 `wx.LEFT` 、 `wx.CENTRE` 、 `wx.RIGHT` ， `vert` 的取值有 `wx.TOP, wx.CENTRE,` 和 `wx.BOTTOM` 。你可以使用 `GetDefaultCellAlignment()` 来得到这个默认的单元格对齐方式，该方法返回一个(`horiz, vert)` 元组。

背景和文本的颜色可以使用 `SetDefaultCellTextColour(colour)` 和 `SetDefaultCellBackgroundColour` 方法来设置。同样， `colour` 参数可以是一个 `wx.Colour` 实例或颜色名。相应的 `get` *方法是 `GetDefaultCellTextColour()` 和 `GetDefaultCellBackgroundColour()` 。最后，你可以使用 `SetDefaultCellFont(font)` 和 `GetDefaultCellFont()` 来处理默认的字体。

使用下面的方法，你可以设置单个单元格的相关属性：

```
GetCellAlignment(row, col)
SetCellAlignment(row, col, horiz, vert)

GetCellBackgroundColour(row, col)
SetCellBackgroundColour(row, col, colour)

GetCellFont(row, col)
SetCellFont(row, col, font)

GetCellTextColour(row, col)
SetCellTextColour(row, col, colour)
```

也可使用 `SetSelectionBackground(colour)` 和 `SetSelectionForeground(colour)` 方法来使用被选的单元格有另外背景色和前景色，相应的 `get` *方法是 `GetSelectionBackground()` 和 `GetSelectionForeground()` 。

你也可以使用 `SetMargins(extraWidth, extraHeight)` 方法来设置网格控件与它的容器的边距。

在内部，类 `wx.grid.Grid` 使用一个名为 `wx.grid.GridCellAttr` 类来管理每个单元格的属性。 `wx.grid.GridCellAttr` 类对于本节所讨论到的属性，也有 `get` 和 *set* 方法。你可以通过使用 `GetOrCreateCellAttr(row, col)` 方法来得到关于一个特定的单元格的 `attr` 对象，它是单元格的属性对象。一个单元格的属性对象仅在该单元格已定义了非默认的属性时才被创建。一旦你有了该单元格的属性对象，你就可以用它来定义该单元格的显示属性。

要创建你自己的单元格属性对象，这个构造函数是 `wx.grid.GridCellAttr()` 。你可以设置某些参数，然后将该对象传递给方法 `SetColAttr(attr)` 或 `SetRowAttr(attr)` ，这两个方法将将这些显示属性应用到该行或列中的每个单元格，如例14.6所示。

如果你在使用一个网格表，你可以覆盖方法 `GetAttr(row, col)` 来返回特定单元格的一个 `wx.grid.GridCellAttr` 实例。

你也可以改变网格线的颜色和显示。网格线的显示是由方法 `EnableGridLines(enable)` 来控制的。参数 `enable` 是一个布乐值。如果为 `True` ，网格线被显示，如果为 `False` ，则不显示。你可以使用方法 `SetGridLineColor(colour)` 来改变网格线的颜色。

自定义描绘器和编辑器

是什么使得网格控件是如此的灵活和有用呢？它就是显示或编辑一个单元格的内容的机制可以被改变这一特性。在后面的几节中，我们将给你展示如何去使用预定义的描绘器和编辑器，以及如何写你自己的描绘器和编辑器。

如何使用一个自定义的单元格描绘器？

默认情况下，网格将它的数据以简单字符串的形式显示，然而，你也可以以不同的格式显示你的数据。你可以想将布尔值数据显示为一个复选框，或以图片格式显示一个数字值，或将一个数据的列表以线条的方式显示。

在 `wxPython` 中，每个单元格都可以有它自己的描绘器，这使得它能够以不同的方式显示它的数据。下面的部分讨论几个 `wxPython` 中预定义的描绘器，以及如何定义你自己的描绘器。

预定义的描绘器**(`renderer`)**

一个网格描绘器是类 `wx.grid.GridCellRenderer` 的一个实例， `wx.grid.GridCellRenderer` 是一个抽象的父类。一般，你会使用它的子类。表14.4说明了几个你可以用在你的单元格中的预定义的描绘器。它们都有一个构造函数和 `get` , *`set`* 方法。

表**14.4** 预定义的网格单元格描绘器

`wx.grid.GridCellAutoWrapStringRenderer`	显示文本化的数据，在单元格边界按词按行。
`wx.grid.GridCellBoolRenderer`	使用一个复选框来描绘布尔数据——选中表示 `True` ，未选中表示 `False` 。
`wx.grid.GridCellDateTimeRenderer`	使单元格能够显示一个格式化的日期或时间。
`wx.grid.GridCellEnumRenderer`	文本形式。
`wx.grid.GridCellFloatRenderer`	使用指定位数和精度来描绘浮点数。该类的构造函数要求两个参数(`width=` -1, `precision=` -1)。默认的对齐方式为右对齐。
`wx.grid.GridCellNumberRenderer`	数字数据。默认为右对齐方式显示。
`wx.grid.GridCellStringRenderer`	简单字符串的形式。

要得到一个特定单元格的描绘器，可以使用方法 `GetCellRenderer(row, col)` ，该方法返回指定坐标处的单元格的描绘器实例。要为一个单元格设置描绘器，可以使用 `SetCellRenderer(row, col, renderer)` 方法，其中 `renderer` 参数是用于指定单元格的新的描绘器。这些方法都简单地设置或得到存储在相关单元格属性对象中的描绘器，所以如果你愿意的话，你可以直接处理 `GridCellAttr` 。你可以通过使 `GetDefaultRenderer` 和 `SetDefaultRenderer(renderer)` 来得到和设置用于整个网格的默认的描绘器。

你也可以为一行设置描绘器，这个的典型应用是电子表格中的某列总是显示特定类型的数据。实现的方法是 `SetColFormatBool(col)`, `SetColFormatNumber(col)` ，以及 `SetColFormatFloat(col, width, precision)` 。

创建一个自定义的描绘器

要创建你自定义的单元格描绘器，需要创建 `wx.grid.PyGridCellRenderer` 的一个子类。创建自定义的单元格描绘器，使你能够以特定的格式显示相关的数据。

图14.6显示了一个自定义描绘器的示例，它随机地绘制单元格的背景色。

图**14.6**

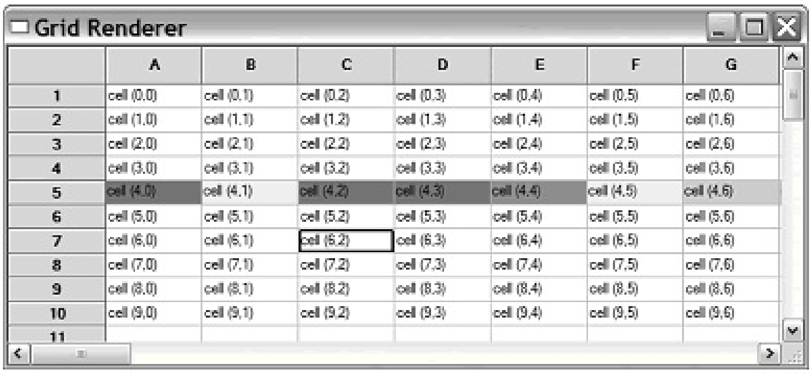


Figure 14.6
A custom grid renderer, setting background colors

例14.7是产生图14.6的代码，其中显示了相关的类和所覆盖的方法

例**14.7**

```
#-*- encoding:UTF-8 -*-
import wx
import wx.grid
import random

class RandomBackgroundRenderer(wx.grid.PyGridCellRenderer):#定义
描绘器
    def __init__(self):
        wx.grid.PyGridCellRenderer.__init__(self)

    def Draw(self, grid, attr, dc, rect, row, col, isSelected):#
绘制
        text = grid.GetCellValue(row, col)
        hAlign, vAlign = attr.GetAlignment()
        dc.SetFont( attr.GetFont() )
        if isSelected:
            bg = grid.GetSelectionBackground()
            fg = grid.GetSelectionForeground()
        else:
            bg = random.choice(["pink", "sky blue", "cyan", "yel
low", "plum"])
            fg = attr.GetTextColour()

        dc.SetTextBackground(bg)
        dc.SetTextForeground(fg)
        dc.SetBrush(wx.Brush(bg, wx.SOLID))
        dc.SetPen(wx.TRANSPARENT_PEN)
dc.DrawRectangleRect(rect)
        grid.DrawTextRectangle(dc, text, rect, hAlign, vAlign)

    def GetBestSize(self, grid, attr, dc, row, col):
        text = grid.GetCellValue(row, col)
        dc.SetFont(attr.GetFont())
        w, h = dc.GetTextExtent(text)
        return wx.Size(w, h)

    def Clone(self):
```

```
        return RandomBackgroundRenderer()

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Renderer",
                          size=(640,480))

        grid = wx.grid.Grid(self)
        grid.CreateGrid(50,50)

        # Set this custom renderer just for row 4
        attr = wx.grid.GridCellAttr()
        attr.SetRenderer(RandomBackgroundRenderer())
        grid.SetRowAttr(4, attr)#赋于第5行

        for row in range(10):
            for col in range(10):
                grid.SetCellValue(row, col,
                                  "cell (%d,%d)" % (row, col))

app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

你的描绘器类必须覆盖基类的下面三个方法。

```
Draw()
GetBestSize()
Clone()
```

这三个方法中最重要的是 `Draw(grid, attr, dc, rect, row, col, isSelected)` 。其中参数 `grid` 是包含相应单元格的网格实例。参数 `attr` 是网格的属性实例。如果你需要使用基本的绘制方法的话，参数 `dc` 是用于绘制的设备上下文。参数 `rect` 是单元格的矩形区域。参数 `row,col` 是单元格的坐标，如果单元格当前处于被选状态的话，参数 `isSelected` 为 `True` 。在你的绘制方法中，你可以自由地做任何你想做的事情。

方法 `GetBestSize(grid, attr, dc, row, col)` 返回一个 `wx.Size` 实例，该实例代表单元格的首先尺寸。方法 `Clone()` 返回一个 `wx.grid.GridCellRenderer` 实例。一旦描绘器被定义了，你就可以像使用预定义的描绘器一样使用它。

如何编辑一个单元格？

wxPython 的网格控件允许你编辑单元格中的值。敲击一个单元格，或开始键入一个新的数据值都将打开一个默认的字符串编辑器，让你可以输入不同的字符串。在这一节，我们将讨论多种修改此默认行为的方法。

你可以使用方法 EnableEditing(enable) 来开关整个网格的可编辑性——参数 enable 是一个布尔值。如果它是 False ，那么所有的单元格都不可编辑。如果关闭了网格的可编辑性，那么你就不能再设置单个单元格的编辑状态了。如果打开了网格的可编辑性的话，单个的单元格可以被指定为只读。你可以使用方法 IsEditable() 来确定网格是否可编辑。

你可以使用方法 SetReadOnly(row, col, isReadOnly=True) 来设置一个特定单元格的编辑状态。 isReadOnly=True 代表该单元格为只读，为 False 代表单元格可编辑。 SetReadOnly() 是类 wx.grid.GridCellAttr 中的同名方法的一个简捷方式。换句话说，你可以使用 GetCellAttr(row, col).SetReadOnly(isReadOnly) 之类的来将一个单元格设置为只读。使用单元格属性机制的好处就是你可以将 SetReadOnly 与 SetRowAttr() 和 SetColAttr() 方法结合起来，以一次性的将整个行或列设置为可编辑的或只读的。

你也可以使用方法 EnableCellEditControl(enable=True) 和 DisableCellEditControl() 来处理网格的可编辑性，第二个方法等同于 EnableCellEditControl(False) 。 Enable 方法将在当前所选择的单元格中创建并显示该单元格的编辑器。 *disable* 方法则相反。如果 enable *方法将工作于当前单元格，那么 CanEnableCellControl() 返回 true ，这就意味该网格是可编辑并且单元格没有被指定为只读。如果当前单元格的编辑器被激活了，则方法 IsCellEditControlEnabled() 返回 true 。

这里还有一些内在的可用的方法，你可以用于对编辑进行更细致的处理。你可以使用方法 ShowCellEditControl() 来触发当前单元格的编辑，并且你也可以使用方法 HideCellEditControl() 该编辑。你可以使用方法 IsCurrentCellReadOnly() 来确定当前单元格可编辑的有效性。你可以使用方法 SaveEditControlValue() 来确保在编辑器中所输入的新值被存储。当焦点从被编辑的单元格上移走时，网格控件隐式地调用该方法，当在你的程序中所做的一些事情可能会导致值被丢失时（比如关闭网格所处的窗口时），隐式地调用该方法是一个好的方式。

每个单元格都有它自己特定的编辑器对象。你可以使用方法 GetCellEditor(row, col) 来得到相关单元格的编辑器的一个引用，返回值是是类 wx.grid.GridCellEditor 的一个实例。你可以使用方法 SetCellEditor(row, col, editor) 来设置该编辑器，其中的 editor 参数是一个 wx.grid.GridCellEditor 。你可以使用方法 GetDefaultEditor() 和 SetDefaultEditor(editor) 来为整个网格管理默认的编辑器。正如描绘器一样，编辑器对象作为与单元格、行或列相关的 wx.grid.GridCellAttr 的一部分被存储。

如何使用一个自定义的单元格编辑器？

正如描绘器一样，`wxPython` 提供了几个不同类型的标准编辑器，也让你可以创建你自己的编辑器。

预定义的编辑器

所有的 `wxPython` 编辑器都是类 `wx.grid.GridCellEditor` 的子类。表14.5说明了这些标准的编辑器。

在接下来的部分，我们将给你展示如何创建自定义的单元格编辑器。

表14.5 `wxPyhton` 中的单元格编辑器

`wx.grid.GridCellAutoWrapStringEditor`	使用多行文本控件来编辑数据值
`wx.grid.GridCellBooleanEditor`	用于单元格布尔值的编辑器，由框构成必，双击显示该复选框。布尔值描绘器用于一个布尔值编你可以用1或0，或者 on / off 东西来替代布尔值描绘器来显示选状态。
`wx.grid.GridCellChoiceEditor`	复合框编辑器。这个构造函数要数
`wx.grid.GridCellEnumEditor`	继承自 `wx.grid.GridCellChoiceE` 将数字换成等同的字符串呈现给
`wx.grid.GridCellFloatEditor`	用于输入指定精度的浮点数。这数要求的参数是(`width=` -1， `precision=` -1) 的意义与相应描绘器中的意思一这个编辑器输入的数被转换到相和精度。
`wx.grid.GridCellNumberEditor`	整数编辑器。该构造函数要求的(`min=` -1, `max=` -1)。如果 `min` 和 `max` 设置了，那么这器将进行范围检查，并否决试图之外的数。单元格的右边会有一个 `spinner` 控件，使用户可以来改变单元格中的值。
`wx.grid.GridCellTextEditor`	默认的文本编辑器。

创建自定义的编辑器

你可以想创建一个自定义的编辑器自行处理输入的数据。要创建你自己的编辑器，你要创建 `wx.grid.PyGridCellEditor` 的一个子类。这比描绘器复杂些。表14.6显示了几个你需要覆盖的方法。

表14.7显示了父类的更多的方法，你可以覆盖它们以改进你的自定义编辑器的外观。

表**14.6** 你必须覆盖的 **`PyGridCellEditor`** 的方法

`BeginEdit(row, col, grid)`	参数 `row,col` 是单元格的坐标， `grid` 是包含的单元格。该方法在编辑请求之初被调用。在该方法中，编辑器用于得到数据去编辑，并为编辑做准工作。
`Clone()`	返回该编辑器的一个拷贝。
`Create(parent, id, evtHandler)`	创建被编辑器使用的控件。参数 `parent` 是容器， `id` 是要创建的控件的标识符， `evtHandler` 是绑定到该新控件的事件处理器。
`EndEdit(row, col, grid)`	如果编辑已经改变了单元格的值，则返回 `True` 。任何其它的必须的清除工作都应该在这里被执行。
`Reset()`	如果编辑被取消了，则该方法被调用。此时应该将控件中的值还原为初始值。

表**14.7** 可以覆盖的 **`PyGridCellEditor`** 的方法

Destroy()	当编辑被销毁时，执行任何最终的清除工作。
IsAcceptedKey(evt)	如果 evt 中的键被按下会启动编辑器，则方法返回 True 。 F2 键始终都用于启动编辑器。蕨类假设任意键的按下都将启动编辑器，除非它被修改为通过 control,alt, 或 shift 来启动。
PaintBackground(rect, attr)	参数 rect 是一个 wx.Rect （使用逻辑单位）， attr 是与单元格相关的 wc.grid.GridCellAttr 。该方法的目的是绘制没有被编辑器控件所覆盖的单元格的部分。基类通过属性得到背景色并使用得到的背景色填充矩形。
SetSize(rect)	rect 是一个该控件在屏幕上的逻辑尺度的 wx.Rect 。如果必要的话，可以使用该方法来将控件定位在该矩形内。
Show(show, attr)	参数 show 是一个布尔值，它决定是否显示编辑器， attr 是相关单元格的属性实例。调用该方法来显示或隐藏编辑器。
StartingClick()	当编辑器通过在单元格上的一个鼠标敲击被启动时，该方法被调用来允许编辑器将该敲击用于自己的目的。
StartingKey(evt)	如果编辑通过一个按键的按压被启动了，那么该方法被调用来允许编辑器控件使用该按键，如果你想的话。（例如，通过使用它作为实际编辑器的一部分）。

一旦你的编辑器完成了，你就可以使用 SetCellEditor 方法将它设置为任何单元格的编辑器。例14.8显示了一个自定义编辑器的示例，这个例子自动将你输入的文本转换为大写。

例**14.8** 创建自定义的大写编辑器

```
#-*- encoding:UTF-8 -*-
import wx
import wx.grid
import string

class UpCaseCellEditor(wx.grid.PyGridCellEditor):#声明编辑器
    def __init__(self):
        wx.grid.PyGridCellEditor.__init__(self)
```

```
    def Create(self, parent, id, evtHandler):#创建
        """
 Called to create the control, which must derive from wx.Control
.
 *Must Override*
 """
        self._tc = wx.TextCtrl(parent, id, "")
        self._tc.SetInsertionPoint(0)
        self.SetControl(self._tc)

        if evtHandler:
            self._tc.PushEventHandler(evtHandler)

        self._tc.Bind(wx.EVT_CHAR, self.OnChar)

    def SetSize(self, rect):
        """
 Called to position/size the edit control within the cell rectan
gle.
 If you don't fill the cell (the rect) then be sure to override
 PaintBackground and do something meaningful there.
 """
        self._tc.SetDimensions(rect.x, rect.y, rect.width+2, rec
t.height+2,
                               wx.SIZE_ALLOW_MINUS_ONE)

    def BeginEdit(self, row, col, grid):
        """
 Fetch the value from the table and prepare the edit control
 to begin editing.  Set the focus to the edit control.
 *Must Override*
 """
        self.startValue = grid.GetTable().GetValue(row, col)
        self._tc.SetValue(self.startValue)
        self._tc.SetInsertionPointEnd()
        self._tc.SetFocus()
        self._tc.SetSelection(0, self._tc.GetLastPosition())

    def EndEdit(self, row, col, grid):
        """
 Complete the editing of the current cell. Returns True if the v
alue
 has changed.  If necessary, the control may be destroyed.
 *Must Override*
 """
        changed = False

        val = self._tc.GetValue()

        if val != self.startValue:
            changed = True
            grid.GetTable().SetValue(row, col, val) # update the
```

```
 table

        self.startValue = ''
        self._tc.SetValue('')
        return changed

    def Reset(self):
        """
 Reset the value in the control back to its starting value.
 *Must Override*
 """
        self._tc.SetValue(self.startValue)
        self._tc.SetInsertionPointEnd()

    def Clone(self):
        """
 Create a new object which is the copy of this one
 *Must Override*
 """
        return UpCaseCellEditor()

    def StartingKey(self, evt):
        """
 If the editor is enabled by pressing keys on the grid, this wil
l be
 called to let the editor do something about that first key if d
esired.
 """
        self.OnChar(evt)
        if evt.GetSkipped():
            self._tc.EmulateKeyPress(evt)

    def OnChar(self, evt):
        key = evt.GetKeyCode()
        if key   255:
            evt.Skip()
            return
        char = chr(key)
        if char in string.letters:
            char = char.upper()
            self._tc.WriteText(char)#转换为大写
        else:
            evt.Skip()

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Grid Editor",
                          size=(640,480))

        grid = wx.grid.Grid(self)
        grid.CreateGrid(50,50)
        grid.SetDefaultEditor(UpCaseCellEditor())#设置成默认编辑器
```

```
app = wx.PySimpleApp()
frame = TestFrame()
frame.Show()
app.MainLoop()
```

捕获用户事件

网格控件有许多的用户事件。我们将把它们分为鼠标事件和键盘事件。

如何捕获用户的鼠标动作？

对于网格控件，除了不同的鼠标事件类型外，对于这些类型还有一些不同的事件类。最常用的事件类是 `wx.grid.GridEvent` 。网格的事件类是 `wx.CommandEvent` 的一个子类，它提供了用于获得事件详情的几个方法，如表14.8所示。

表**14.8** **`wx.grid.GridEvent`** 的方法

`AltDown()`	当事件被触发时，如果 `alt` 键被按下了，则返回 `True` 。
`ControlDown()`	当事件被触发时，如果 `control` 键被按下了，则返回 `True` 。
`GetCol()`	返回发生事件的单元格所在的列的索引。
`GetPosition()`	返回返回一个 `wx.Point` 。它代表事件发生点的逻辑坐标（以像素为单位）。
`GetRow()`	返回发生事件的单元格所在的行的索引。
`MetaDown()`	当事件被触发时，如果 `met` 键被按下了，则返回 `True` 。
`Selecting()`	如果事件是一个被选事件，则返回 `True` ，如果事件是取消选择事件，则返回 `False` 。
`ShiftDown()`	当事件被触发时，如果 `shift` 键被按下了，则返回 `True` 。

与 `wx.grid.GridEvent` 相关的有几个不同的事件类型。如表14.9所示。

表**14.9** 关于网格鼠标事件的单元格事件类型

`wx.grid.EVT_GRID_CELL_CHANGE`	当用户通过编辑器改变单元格中的数据时触发该事件。
`wx.grid.EVT_GRID_CELL_LEFT_CLICK`	当用户在一个单元格中敲击鼠标左键时触发该事件。
`wx.grid.EVT_GRID_CELL_LEFT_DCLICK`	当用户在一个单元格中双击鼠标左键时触发该事件。
`wx.grid.EVT_GRID_CELL_RIGHT_CLICK`	当用户在一个单元格中敲击鼠标右键时触发该事件。
`wx.grid.EVT_GRID_CELL_RIGHT_DCLICK`	当用户在一个单元格中双击鼠标右键时触发该事件。
`wx.grid.EVT_GRID_EDITOR_HIDDEN`	当在编辑会话结束时隐藏一个单元格编辑器则触发该事件。
`wx.grid.EVT_GRID_EDITOR_SHOWN`	当在编辑会话结束时显示一个单元格编辑器则触发该事件。
`wx.grid.EVT_GRID_LABEL_LEFT_CLICK`	当用户在行或列的标签区域敲击鼠标左键时触发该事件。
`wx.grid.EVT_GRID_LABEL_LEFT_DCLICK`	当用户在行或列的标签区域双击鼠标左键时触发该事件。
`wx.grid.EVT_GRID_LABEL_RIGHT_CLICK`	当用户在行或列的标签区域敲击鼠标右键时触发该事件。
`wx.grid.EVT_GRID_LABEL_RIGHT_DCLICK`	当用户在行或列的标签区域双击鼠标右键时触发该事件。
`wx.grid.EVT_GRID_Select_CELL`	当用户将焦点移到一个新的单元格，并选择它时触发该事件。

有两个事件类型，它们有一个 `wx.grid.GridSizeEvent` 实例。这两个事件类型分别是 `wx.grid.EVT_GRID_COL_SIZE` ：当列大小被改变时触发， `wx.grid.EVT_GRID_ROW_SIZE` ：当行的大小被改变时触发。网格的尺寸事件有5个与 `wx.GridEvent` ！`AltDown(), ControlDown(), GetPosition(), MetaDow(),` 和 `ShiftDown` 相同的方法。 `wx.grid.GridSizeEvent` 的最后的一个方法是 `GetRowOrCol()` ，该方法返回发生改变的列或行的索引，当然这依赖于具体的事件类型。

事件类型 `wx.grid.EVT_GRID_RANGE_Select` 有一个 `wx.grid.GridRangeSelectEvent` 的实例，该事件当用户选择连续矩形范围内的单元格中被触发。该事件的实例拥有的方法是 `GetBottomRightCoords(), GetBottomRow(), GetLeftCol(), GetRightCol(` 和 `GetTopRow()` ，它们返回被选择区域的整数索引或(`row, col)` 元组。

最后，事件类型 `EVT_GRID_EDITOR_CreateD` 有一个 `wx.grid.GridEditorCreatedEvent` 实例。这个事件在当通过一个编辑会话创建了一个编辑器时被触发。该事件实例的方法有 `GetCol()`, `GetRow()`, 和 `GetControl()` ，它们分别返回发生事件的列，行的索引和使用的编辑控件。

如何捕获用户的键盘动作？

除了使用鼠标外，用户还可以使用键盘来在网格中移动。你可以通过代码的方法来使用表14.10中的移动方法改变光标。其中的许多方法都要求一个 `expandSelection` 参数。每个方法中的 `expandSelection` 的作用都相同。如果这个参数为 `True` ，那么当前的选项将被扩展以容纳这个新的光标位置。如果这个参数为 `False` ，那么当前的选项被新的光标所取代。

表**14.10** 网格光标移动方法

`MoveCursorDown(expandSelection)`	向下移动光标。如果 `expandSelection` 为 `Fal` 等同于按下"下箭头键"，如果 为 `True` ，则等同于按下" `sh` 下箭头键"。
`MoveCursorDownBlock(expandSelection)`	向下移动光标。如果 expandSelection 为 Fal 则等同于" ctrl -下箭头键"， 为 True ，则等同于" shift - ctrl -下箭头键"
`MoveCursorLeft(expandSelection)`	向左移动光标。如果 `expandSelection` 为 `Fal` 等同于按下"左箭头键"，如果 为 `True` ，则等同于按下" `sh` 左箭头键"。
`MoveCursorLeftBlock(expandSelection)`	向左移动光标。如果 expandSelection 为 Fal 则等同于" ctrl -左箭头键"， 为 True ，则等同于" shift - ctrl -左箭头键"
`MoveCursorRight(expandSelection)`	向右移动光标。如果 `expandSelection` 为 `Fal` 等同于按下"右箭头键"，如果 为 `True` ，则等同于按下" `sh` 右箭头键"。
	向右移动光标。如果 expandSelection 为 Fal

MoveCursorRightBlock(expandSelection)	则等同于" ctrl -右箭头键"， 为 True ，则等同 于" shift - ctrl -右箭头键"
MoveCursorUp(expandSelection)	向上移动光标。如 果 expandSelection 为 Fal 等同于按下"上箭头键"，如果 为 True ，则等同于按下" shi 上箭头键"。
MoveCursorUpBlock(expandSelection)	向上移动光标。如 果 expandSelection 为 Fal 则等同于" ctrl -上箭头键"， 为 True ，则等同 于" shift - ctrl -上箭头键"
MovePageDown()	显示下一页的单元格。
MovePageUp()	显示上一页的单元格。

我们已经涵盖了所有你需要了解的有关单元格的知识。在下一章中，我们将讨论树形控件。

本章小结

1、网格控件使你能够创建像电子表格一样的网格表，并具有很大的可控性和灵活性。网格控件是类 `wx.grid.Grid` 的一个实例。通常，如果使用网格控件处理复杂的问题的话，你应该通过 `__init__` 方法来定义它的子类，这是值得的，而非仅仅创建基类的一个实例并在程序的其它地方调用它的方法。

2、有两种方法用来将数据放入一个网格控件中。网格控件可以使用 `CreateGrid(numRows, numCols)` 方法被显式创建，然后使用 `SetCellValue(row, col, s)` 方法来设置单个的单元格。另一种是，你可以创建一个网格表的实例，该网格表作为网格的一个模型，它使你可以很容易地使用另一数据源的数据并显示在网格中。网格表是 `wx.grid.PyGridTableBase` 的子类， `wx.grid.PyGridTableBase` 的方法中， `GetValue(row, col)` 可以被覆盖以在显示一个单元格时驱动网格的行为。网格表被连接到网格控件使用方法 `SetTable(table)` 。当使用网格表的方法创建了网格后，可以通过网格表的方法来改变网格的行和列数。

3、网格也有行和列标签，标签有默认的值，类似于电子表格。标签所显示的文本和标签的其它显示属性可以使用网格的方法来改变。每个项的行和列的尺寸可以被显式了设置，或者网格可以根据所显示的自动调整尺寸。用户也可通过拖动网格线来改变网格的尺寸。如果需要的话，你可以为每行或每列设置一个最小的尺寸，以防止单元格变得太小而不能显示相应的数据。另外，特定的单元格了能使用 `SetCellSize(row, col, numrows, numcols)` 方法来达到跨行或列的目的。

4、用户可以选择网格中的一个或多个矩形范围的网格，这也可以通过使用很多不同的 `select` *方法以程序化的方式实现相同的效果。一个没有在显示区域中的网格单元，可能使用 `MakeCellVisible(row, col)` 方法来将它移到显示区域上。

5、网格控件的强大性和灵活性来源于可以为每个单元格创建自定义的描绘器和编辑器这一能力。描绘器用于控件单元格中的信息显示。默认的描绘器只是一个简单的字符串，但是还有用于布尔值、整数和浮点数的预先定义好（预定义）的描绘器。你可以通过子类化 `wx.Grid.PyGridCellRenderer` 创建你自己的描绘器并覆盖它的绘制方法。

6、默认情况下，网格允许就地编辑数据。你可以改变这个属性（针对单元格，或行或列，或整个网格）。当编辑时，编辑器对象控制显示给用户的东西。默认的编辑器是一个用以修改字符串的普通的文本编辑器控件。其它还有用于布尔值、整数和浮点数的预定义的编辑器。你可以通过子类化 `wx.grid.GridCellEditor` 并覆盖它的几个方法来创建自己的自定义的编辑器。

7、网格控件有许多你能捕获的不同的事件，分别包括单元格中的鼠标敲击和标签中的鼠标敲击事件，以及通过改变一个单元格的尺寸而触发的事件。另外，你能够以编程的方式在网格中移动光标。

第十五章 树形控件

本章内容

- 创建树形控件并添加项目
- 使用样式来设计树形控件
- 在程序中访问树形控件
- 处理树形控件中的选择
- 控制项目的可见性

树形控件是用于显示复杂数据的控件。这里，树形控件被设计用来通过分级层来显示数据，你可以看到每 块数据都有父子方面的东西。一个标准的例子就是文件树，其中的目录中有子目录或文件，从而形成了文 件的一个嵌套的层次。另一个例子是 `HTML` 或 `XML` 文档的文档对象模型(`DOM)` 树。和列表与网格控件一样，树 形控件也提供了在项目显示方面的灵活性，并允许你就地编辑树形控件中的项目。在这一章中，我们将给 你展示如何编辑树形控件中的项目及如何响应用户事件。

创建树形控件并添加项目

树形控件是类 `wx.TreeCtrl` 的实例。图15.1显示了一个树形控件的样例。

图**15.1**

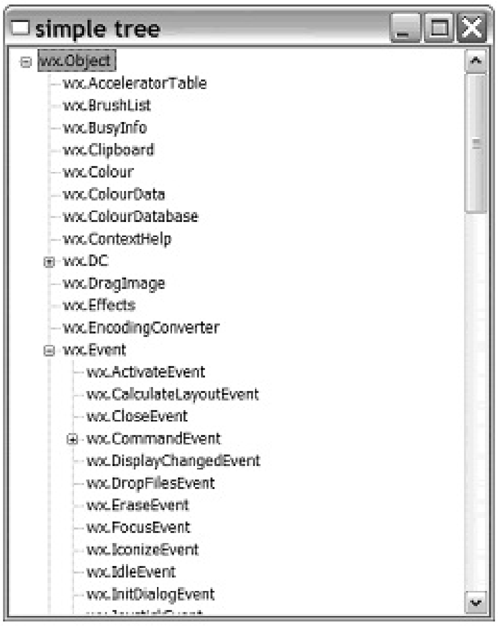


Figure 15.1 A basic tree control example

例15.1是产生图15.1的代码。这个例子中的机制我们将在后面的部分讨论。注意其中的树形控件中的数据 是来自于一个名为 `data.py` 外部文件的。

例**15.1** 树形控件示例

```
import wx
import data
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="simple tree", size=
(400,500))
        # Create the tree
        self.tree = wx.TreeCtrl(self)
        # Add a root node
        root = self.tree.AddRoot("wx.Object")
        # Add nodes from our data set
        self.AddTreeNodes(root, data.tree)
        # Bind some interesting events
        self.Bind(wx.EVT_TREE_ITEM_EXPANDED, self.OnItemExpanded
, self.tree)
        self.Bind(wx.EVT_TREE_ITEM_COLLAPSED, self.OnItemCollaps
ed, self.tree)
        self.Bind(wx.EVT_TREE_SEL_CHANGED, self.OnSelChanged, se
lf.tree)
        self.Bind(wx.EVT_TREE_ITEM_ACTIVATED, self.OnActivated,
self.tree)
        # Expand the first level
        self.tree.Expand(root)
    def AddTreeNodes(self, parentItem, items):
        """
 Recursively traverses the data structure, adding tree nodes to
 match it.
 """
        for item in items:
            if type(item) == str:
                self.tree.AppendItem(parentItem, item)
            else:
                newItem = self.tree.AppendItem(parentItem, item[
0])
                self.AddTreeNodes(newItem, item[1])

    def GetItemText(self, item):
        if item:
            return self.tree.GetItemText(item)
        else:
            return ""

    def OnItemExpanded(self, evt):
        print "OnItemExpanded: ", self.GetItemText(evt.GetItem()
)

    def OnItemCollapsed(self, evt):
        print "OnItemCollapsed:", self.GetItemText(evt.GetItem()
)
    def OnSelChanged(self, evt):
        print "OnSelChanged: ", self.GetItemText(evt.GetItem())
    def OnActivated(self, evt):
```

```
        print "OnActivated: ", self.GetItemText(evt.GetItem())
app = wx.PySimpleApp(redirect=True)
frame = TestFrame()
frame.Show()
app.MainLoop()
```

下面的 `wx.TreeCtrl` 的构造函数是一个典型的 `wxPython` 窗口部件构造函数：

```
wx.TreeControl(parent, id=-1, pos=wx.DefaultPosition,
        size=wx.DefaultSize, style=wx.TR_HAS_BUTTONS,
        validator=wx.DefaultValidator, name="treeCtrl")
```

其中的参数意义与通常的 `wx.Window` 对象相同。该构造函数提供给你了一个没有元素的空的树。

另附 **`data.py`** 文件：

```
# Some sample data for the treectrl samples
tree = [
    "wx.AcceleratorTable",
    "wx.BrushList",
    "wx.BusyInfo",
    "wx.Clipboard",
    "wx.Colour",
    "wx.ColourData",
    "wx.ColourDatabase",
    "wx.ContextHelp",
    ["wx.DC", [
          "wx.ClientDC",
          ["wx.MemoryDC", [
                "wx.lib.colourchooser.canvas.BitmapBuffer",
                ["wx.BufferedDC", [
                      "wx.BufferedPaintDC", ]]]],
          "wx.MetaFileDC",
          "wx.MirrorDC",
          "wx.PaintDC",
          "wx.PostScriptDC",
          "wx.PrinterDC",
          "wx.ScreenDC",
          "wx.WindowDC",]],
    "wx.DragImage",
    "wx.Effects",
    "wx.EncodingConverter",
    ["wx.Event", [
          "wx.ActivateEvent",
          "wx.CalculateLayoutEvent",
          "wx.CloseEvent",
          ["wx.CommandEvent", [
                "wx.calendar.CalendarEvent",
                "wx.ChildFocusEvent",
```

```
                "wx.ContextMenuEvent",
                "wx.gizmos.DynamicSashSplitEvent",
                "wx.gizmos.DynamicSashUnifyEvent",
                "wx.FindDialogEvent",
                "wx.grid.GridEditorCreatedEvent",
                "wx.HelpEvent",
                ["wx.NotifyEvent",[
                      ["wx.BookCtrlEvent", [
                            "wx.ListbookEvent",
                            "wx.NotebookEvent ",]],
                      "wx.grid.GridEvent",
                      "wx.grid.GridRangeSelectEvent",
                      "wx.grid.GridSizeEvent",
                      "wx.ListEvent",
                      "wx.SpinEvent",
                      "wx.SplitterEvent",
                      "wx.TreeEvent",
                      "wx.wizard.WizardEvent ",]],
                ["wx.PyCommandEvent", [
                      "wx.lib.colourselect.ColourSelectEvent",
                      "wx.lib.buttons.GenButtonEvent",
                      "wx.lib.gridmovers.GridColMoveEvent",
                      "wx.lib.gridmovers.GridRowMoveEvent",
                      "wx.lib.intctrl.IntUpdatedEvent",
                      "wx.lib.masked.combobox.MaskedComboBoxSele
ctEvent",
                      "wx.lib.masked.numctrl.NumberUpdatedEvent"
,
                      "wx.lib.masked.timectrl.TimeUpdatedEvent "
,]],
                "wx.SashEvent",
                "wx.ScrollEvent",
                "wx.stc.StyledTextEvent",
                "wx.TextUrlEvent",
                "wx.UpdateUIEvent",
                "wx.WindowCreateEvent",
                "wx.WindowDestroyEvent ",]],
          "wx.DisplayChangedEvent",
          "wx.DropFilesEvent",
          "wx.EraseEvent",
          "wx.FocusEvent",
          "wx.IconizeEvent",
          "wx.IdleEvent",
          "wx.InitDialogEvent",
          "wx.JoystickEvent",
          "wx.KeyEvent",
          "wx.MaximizeEvent",
          "wx.MenuEvent",
          "wx.MouseCaptureChangedEvent",
          "wx.MouseEvent",
          "wx.MoveEvent",
          "wx.NavigationKeyEvent",
          "wx.NcPaintEvent",
```

```
            "wx.PaintEvent",
            "wx.PaletteChangedEvent",
            "wx.ProcessEvent",
            ["wx.PyEvent", [
                  "wx.lib.throbber.UpdateThrobberEvent ",]],
            "wx.QueryLayoutInfoEvent",
            "wx.QueryNewPaletteEvent",
            "wx.ScrollWinEvent",
            "wx.SetCursorEvent",
            "wx.ShowEvent",
            "wx.SizeEvent",
            "wx.SysColourChangedEvent",
            "wx.TaskBarIconEvent",
            "wx.TimerEvent ",]],
      ["wx.EvtHandler", [
            "wx.lib.gridmovers.GridColMover",
            "wx.lib.gridmovers.GridRowMover",
            "wx.html.HtmlHelpController",
            "wx.Menu",
            "wx.Process",
            ["wx.PyApp", [
                  ["wx.App", [
                        "wx.py.PyAlaCarte.App",
                        "wx.py.PyAlaMode.App",
                        "wx.py.PyAlaModeTest.App",
                        "wx.py.PyCrust.App",
                        "wx.py.PyShell.App",
                        ["wx.py.filling.App", [
                              "wx.py.PyFilling.App ",]],
                        ["wx.PySimpleApp", [
                              "wx.lib.masked.maskededit.test",]],
                        "wx.PyWidgetTester ",]]]],

            "wx.TaskBarIcon",
            ["wx.Timer", [
                  "wx.PyTimer ",]],
            ["wx.Validator", [
                  ["wx.PyValidator",[
                        "wx.lib.intctrl.IntValidator",]]]],
            ["wx.Window", [
                  ["wx.lib.colourchooser.canvas.Canvas", [
                        "wx.lib.colourchooser.pycolourslider.PyCol
ourSlider",
                        "wx.lib.colourchooser.pypalette.PyPalette"
,]],
                  "wx.lib.gridmovers.ColDragWindow",
                  ["wx.Control",[
                        ["wx.BookCtrl", [
                              "wx.Listbook",
                              ["wx.Notebook",[
                                    "wx.py.editor.EditorNotebook",
                                    "wx.py.editor.EditorShellNoteb
ook",]] ]],
```

```
                      ["wx.Button", [
                            ["wx.BitmapButton",[
                                  "wx.lib.colourselect.ColourSel
ect",
                                  "wx.ContextHelpButton",
                                  "wx.lib.foldmenu.FoldOutMenu "
,]] ]],
                      "wx.calendar.CalendarCtrl",
                      "wx.CheckBox",
                      ["wx.ComboBox",[
                            ["wx.lib.masked.combobox.BaseMaskedC
omboBox", [
                                  "wx.lib.masked.combobox.ComboB
ox",
                                  "wx.lib.masked.combobox.PreMas
kedComboBox",]] ]],
                      ["wx.ControlWithItems", [
                            ["wx.Choice",[
                                  "wx.DirFilterListCtrl ",]],
                            "wx.ListBox",
                                  "wx.CheckListBox ",]],
                      "wx.Gauge",
                      "wx.GenericDirCtrl",
                      "wx.gizmos.LEDNumberCtrl",
                      ["wx.ListCtrl",[
                            "wx.ListView ",]],
                      ["wx.PyControl",[
                            "wx.lib.calendar.Calendar",
                            ["wx.lib.buttons.GenButton",[
                                  ["wx.lib.buttons.GenBitmapButt
on",[
                                        ["wx.lib.buttons.GenBitm
apTextButton",[
                                              "wx.lib.buttons.Ge
nBitmapTextToggleButton",]],
                                        "wx.lib.buttons.GenBitma
pToggleButton ",]],
                                  "wx.lib.buttons.GenToggleButto
n ",]],
                            "wx.lib.statbmp.GenStaticBitmap",
                            "wx.lib.stattext.GenStaticText",
                            "wx.lib.popupctl.PopButton",
                            "wx.lib.popupctl.PopupControl",
                            "wx.lib.ticker.Ticker ",]],
                      "wx.RadioBox",
                      "wx.RadioButton",
                      "wx.ScrollBar",
                      "wx.Slider",
                      "wx.SpinButton",
                      "wx.SpinCtrl",
                      ["wx.StaticBitmap",[
                            "wx.lib.fancytext.StaticFancyText ",
]],
```

```
                        "wx.StaticBox",
                        "wx.StaticLine",
                        "wx.StaticText",
                        ["wx.stc.StyledTextCtrl",[
                              ["wx.py.editwindow.EditWindow",[
                                    "wx.py.crust.Display",
                                    "wx.py.editor.EditWindow",
                                    "wx.py.filling.FillingText",
                                    "wx.py.shell.Shell",]],
                              "wx.lib.pyshell.PyShellWindow ",]],
                        ["wx.TextCtrl", [
                              ["wx.lib.masked.textctrl.BaseMaskedT
extCtrl",[
                                    "wx.lib.masked.ipaddrctrl.IpAd
drCtrl",
                                    "wx.lib.masked.numctrl.NumCtrl
",
                                    "wx.lib.masked.textctrl.PreMas
kedTextCtrl",
                                    "wx.lib.masked.textctrl.TextCt
rl",
                                    "wx.lib.masked.timectrl.TimeCt
rl ",]],
                              "wx.py.crust.Calltip",
                              "wx.lib.sheet.CTextCellEditor",
                              "wx.py.crust.DispatcherListing",
                              "wx.lib.intctrl.IntCtrl",
                              "wx.lib.rightalign.RightTextCtrl",
                              "wx.py.crust.SessionListing",]],
                        "wx.ToggleButton",
                        "wx.ToolBar",
                        ["wx.TreeCtrl",[
                              "wx.py.filling.FillingTree",
                              "wx.gizmos.RemotelyScrolledTreeCtrl
",]],
                        "wx.gizmos.TreeListCtrl ",]],
                  "wx.gizmos.DynamicSashWindow",
                  "wx.lib.multisash.EmptyChild",
                  "wx.glcanvas.GLCanvas",
                  "wx.lib.imagebrowser.ImageView",
                  "wx.MDIClientWindow",
                  "wx.MenuBar",
                  "wx.lib.multisash.MultiClient",
                  "wx.lib.multisash.MultiCloser",
                  "wx.lib.multisash.MultiCreator",
                  "wx.lib.multisash.MultiSash",
                  "wx.lib.multisash.MultiSizer",
                  "wx.lib.multisash.MultiSplit",
                  "wx.lib.multisash.MultiViewLeaf",
                  ["wx.Panel",[
                        "wx.gizmos.EditableListBox",
                        ["wx.lib.filebrowsebutton.FileBrowseButton
",[
```

```
                            "wx.lib.filebrowsebutton.DirBrowseBu
tton",
                            "wx.lib.filebrowsebutton.FileBrowseB
uttonWithHistory",]],
                      "wx.lib.floatcanvas.FloatCanvas.FloatCanva
s",
                      "wx.lib.floatcanvas.NavCanvas.NavCanvas",
                      "wx.NotebookPage",
                      ["wx.PreviewControlBar",[
                            "wx.PyPreviewControlBar ",]],
                      "wx.lib.colourchooser.pycolourbox.PyColour
Box",
                      "wx.lib.colourchooser.pycolourchooser.PyCo
lourChooser",
                      ["wx.PyPanel",[
                            "wx.lib.throbber.Throbber",]],
                      "wx.lib.shell.PyShell",
                      "wx.lib.shell.PyShellInput",
                      "wx.lib.shell.PyShellOutput",
                      ["wx.ScrolledWindow",[
                            "wx.lib.editor.editor.Editor",
                            ["wx.grid.Grid",[
                                  "wx.lib.sheet.CSheet ",]],
                            ["wx.html.HtmlWindow",[
                                  "wx.lib.ClickableHtmlWindow.Py
ClickableHtmlWindow",]],
                            "wx.PreviewCanvas",
                            "wx.lib.printout.PrintTableDraw",
                            ["wx.PyScrolledWindow",[
                                  "wx.lib.scrolledpanel.Scrolled
Panel",]],
                            "wx.lib.ogl.ShapeCanvas",
                            "wx.gizmos.SplitterScrolledWindow ",
]],
                      ["wx.VScrolledWindow",[
                            ["wx.VListBox", [
                                  "wx.HtmlListBox ",]] ]],
                      ["wx.wizard.WizardPage", [
                            "wx.wizard.PyWizardPage",
                            "wx.wizard.WizardPageSimple ",]],
                "wx.lib.plot.PlotCanvas",
                "wx.lib.wxPlotCanvas.PlotCanvas",
                ["wx.PopupWindow",[
                      "wx.lib.foldmenu.FoldOutWindow",
                      ["wx.PopupTransientWindow",[
                            "wx.TipWindow ",]] ]],
                ["wx.PyWindow", [
                      "wx.lib.analogclock.AnalogClockWindow",]],
                "wx.lib.gridmovers.RowDragWindow",
                ["wx.SashWindow",[
                      "wx.SashLayoutWindow ",]],
                "wx.SplashScreenWindow",
                ["wx.SplitterWindow",[
```

```
                        "wx.py.crust.Crust",
                        "wx.py.filling.Filling",
                        "wx.gizmos.ThinSplitterWindow ",]],
                  "wx.StatusBar",
                  ["wx.TopLevelWindow",[
                        ["wx.Dialog",[
                              "wx.lib.calendar.CalenDlg",
                              "wx.ColourDialog",
                              "wx.DirDialog",
                              "wx.FileDialog",
                              "wx.FindReplaceDialog",
                              "wx.FontDialog",
                              "wx.lib.imagebrowser.ImageDialog",
                              "wx.MessageDialog",
                              "wx.MultiChoiceDialog",
                              "wx.lib.dialogs.MultipleChoiceDialog
",
                              "wx.PageSetupDialog",
                              "wx.lib.popupctl.PopupDialog",
                              "wx.PrintDialog",
                              "wx.lib.dialogs.ScrolledMessageDialo
g",
                              "wx.SingleChoiceDialog",
                              "wx.TextEntryDialog",
                              "wx.wizard.Wizard ",]],
                        ["wx.Frame", [
                              "wx.lib.analogclockopts.ACCustomizat
ionFrame",
                              "wx.py.filling.FillingFrame",
                              ["wx.py.frame.Frame",[
                                    "wx.py.crust.CrustFrame",
                                    ["wx.py.editor.EditorFrame",[
                                          "wx.py.editor.EditorNote
bookFrame",]],
                                    "wx.py.shell.ShellFrame",]],
                              "wx.html.HtmlHelpFrame",
                              "wx.MDIChildFrame",
                              "wx.MDIParentFrame",
                              "wx.MiniFrame",
                              ["wx.PreviewFrame",[
                                    "wx.PyPreviewFrame ",]],
                              "wx.ProgressDialog",
                              "wx.SplashScreen",
                              "wx.lib.splashscreen.SplashScreen",
                              "wx.lib.masked.maskededit.test2",
                              "wx.lib.plot.TestFrame ",]] ]],
                  "wx.gizmos.TreeCompanionWindow ",]] ]] ]],
    "wx.FileHistory",
    "wx.FileSystem",
    "wx.FindReplaceData",
    "wx.FontData",
    "wx.FontList",
    "wx.FSFile",
```

```
    ["wx.GDIObject",[
          "wx.Bitmap",
          "wx.Brush",
          "wx.Cursor",
          "wx.Font",
          "wx.Icon",
          "wx.Palette",
          "wx.Pen",
          "wx.Region ",]],
    "wx.glcanvas.GLContext",
    ["wx.grid.GridTableBase", [
          "wx.grid.GridStringTable",
          "wx.grid.PyGridTableBase ",]],
    ["wx.html.HtmlCell", [
          "wx.html.HtmlColourCell",
          "wx.html.HtmlContainerCell",
          "wx.html.HtmlFontCell",
          "wx.html.HtmlWidgetCell",
          "wx.html.HtmlWordCell ",]],
    "wx.html.HtmlDCRenderer",
    "wx.html.HtmlEasyPrinting",
    "wx.html.HtmlFilter",
    "wx.html.HtmlLinkInfo",
    ["wx.html.HtmlParser", [
          "wx.html.HtmlWinParser ",]],
    "wx.html.HtmlTag",
    ["wx.html.HtmlTagHandler", [
          ["wx.html.HtmlWinTagHandler", [
                "wx.lib.wxpTag.wxpTagHandler ",]] ]],
    "wx.Image",
    ["wx.ImageHandler", [
          ["wx.BMPHandler", [
                ["wx.ICOHandler", [
                      ["wx.CURHandler", [
                            "wx.ANIHandler ",]] ]] ]],
          "wx.GIFHandler",
          "wx.JPEGHandler",
          "wx.PCXHandler",
          "wx.PNGHandler",
          "wx.PNMHandler",
          "wx.TIFFHandler",
          "wx.XPMHandler ",]],
    "wx.ImageList",
    "wx.IndividualLayoutConstraint",
    "wx.LayoutAlgorithm",
    ["wx.LayoutConstraints", [
          "wx.lib.anchors.LayoutAnchors",
          "wx.lib.layoutf.Layoutf",]],
    "wx.ListItem",
    "wx.Mask",
    "wx.MenuItem",
    "wx.MetaFile",
    "wx.PageSetupDialogData",
```

```
        "wx.PenList",
        "wx.PrintData",
        "wx.PrintDialogData",
        "wx.Printer",
        ["wx.Printout", [
              "wx.html.HtmlPrintout",
              "wx.lib.plot.PlotPrintout",
              "wx.lib.printout.SetPrintout ",]],
        ["wx.PrintPreview", [
              "wx.PyPrintPreview ",]],
        "wx.RegionIterator",
        ["wx.Sizer", [
              "wx.BookCtrlSizer",
              ["wx.BoxSizer", [
                    "wx.StaticBoxSizer", ]],
              ["wx.GridSizer", [
                    ["wx.FlexGridSizer", [
                          "wx.GridBagSizer",]] ]],
              "wx.NotebookSizer",
              "wx.PySizer",]],
        ["wx.SizerItem", [
              "wx.GBSizerItem",]],
        "wx.SystemOptions",
        "wx.ToolBarToolBase",
        "wx.ToolTip",
        "wx.gizmos.TreeListColumnInfo",
        "wx.xrc.XmlDocument",
        "wx.xrc.XmlResource",
        "wx.xrc.XmlResourceHandler ",
  ]
```

如何添加一个**root(**根**)**元素？

当你将项目添加到树时，你首先必须要添加的项目是 `root(` 根)元素。添加根元素的方法如下所示：

```
AddRoot(text, image=-1, selImage=-1, data=None)
```

你只能添加一个根元素。如果你在已经存在一个根元素后，再添加第二根元素的话，那么 `wxPython` 将引发一个异常。其中的参数 `text` 包含用于根元素的显示字符串。参数 `image` 是图像列表中的一个索引，代表要显示在参数 `text` 旁边的图像。这个图像列表将在15.5节中作更详细的讨论，但是现在只要知道它的行为类似于用于列表控件的图像列表。参数 `data` 是一个与项目相关的数据对象，主要的目的是分类。

`AddRoot()` 方法返回一个关于根项目的 `ID` 。树形控件使用它自己的类 `wx.TreeItemId` 来管理项目。在大多数时候，你不需要关心 `ID` 的具体值，你只需要知道每个项目都有一个唯一的 `wx.TreeItemId` 就够了，并且这个值可以使

用等号测试。 `wx.TreeItemId` 不映射到任何简单的类型——它的值没有任何的关联性，因为你只是把它用于相等测试。

如何将更多的项目添加到树中？

一旦你有了根元素，你就可以开始向树中添加元素了。用的最多的方法是 `AppendItem(parent, text, image=` -1, `selImage=` -1, `data=None)` 。参数 `parent` 是已有的树项目的 `wx.TreeItemId` ，它作为新项目的父亲。参数 `text` 是显示新项目的文本字符串。参数 `image` 和 `selImage` 的意义与方法 `AddRoot()` 中的相同。该方法将新的项目放置到其父项目的孩子列表的末尾。这个方法返回新创建的项目的 `wx.TreeItemId` 。如果你想给新的项目添加子项目的话，你需要拥有这个 `ID` 。一个示例如下：

```
rootId = tree.AddRoot("The Root")
childId = tree.AppendItem(rootId, "A Child")
grandChildId = tree.AppendItem(childId, "A Grandchild")
```

上面的这个代码片断增加了一个 `root(` 根)项目，然后给根项目添加了一个子项目，然后给子项目添加了它的子项目。

如果要将子项目添加到孩子列表的开头的话，使用方法 `PrependItem(parent, text, image=` -1, `selImage=` -1, `data=None)` 。

如果你想将一个项目插入树的任意点上，你可以使用后面的两种方法之一。第一个是 `InsertItem(parent, previous, text,image=` -1, `selImage=` -1, `data=None)` 。其中参数 `previous` 其父项目中的子列表中的项目的 `wx.TreeItemId` 。插入的项目将放置在该项目的后面。第二个方法是 `InsertItemBefore(parent, before, text, image=` -1, `selImage=` -1, `data=None)` 。该方法将新的项目放置在 `before` 所代表的项目之前。然而参数 `before` 不是一个项目的 `ID` 。它是项目在孩子列表中的整数索引。第二个方法返回新项目的一个 `wx.TreeItemId` 。

如何管理项目？

要去掉树中的一个项目，可以使用 `Delete(item)` 方法，其中参数 `item` 是该项目的 `wx.TreeItemId` 。调用这个方法将导致一个 `EVT_TREE_Delete_ITEM` 类型的树的事件被触发。后面的章节我们将讨论树的事件类型。要删除一个项目的所有子项目，而留下该项目自身，可以使用方法 `DeleteChildren(item)` ，其中参数 `item` 也是一个 `wx.TreeItemId` 。该方法不产生一个删除事件。要清除整个树，使用方法 `DeleteAllItems()` 。该方法为每个项目生成一个删除事件，但是，这在某些老版的 `Windows` 系统上不工作。

一旦你将一个项目添加到树中，你就可以使用方法 `GetItemText(item)` 来得到该项目在显示在树中的文本，其中参数 `item` 是一个 `wx.TreeItemId` 。如果你想改变一个项目的显示文本，可以使用方法 `SetItemText(item, text)` ，其中参数 `item` 是一个 `wx.TreeItemId` ，参数 `text` 是一个新的显示文本。

最后，你可以使用方法 `GetCount()` 来得到树中项目的总数。如果你想得到特定项目下的子项目的数量，可以使用方法 `GetChildrenCount(item, recursively=True)` 。其中参数 `item` 是一个 `wx.TreeItemId` ，参数 `recursively` 如果为 `False` ，那么该方法只返回直接的子项目的数量，如果为 `True` ，则返回所有的子项目而不关嵌套有多深。

树控件的显示样式

树控件的显示样式分为四类。第一类定义了树中显示在父项目的文本旁的按钮（用以展开或折叠子项目）。它们显示在表15.1中。

表**15.1** 树控件中的按钮

`wx.TR_HAS_BUTTONS`	按钮。在 `Windows` 上，+用于标明项目可以被展开，-表示可以折叠。
`wx.TR_NO_BUTTONS`	没有按钮。

接下来的一类显示在表15.2中，它们决定树控件将连接线绘制在何处。

表**15.2** 树控件中的连接线

`wx.TR_LINES_AT_ROOT`	如果设置了这个样式，那么树控件将在多个 `root` 项目之间绘制连线。注意，如果 `wx.TR_HIDE_ROOT` 被设置了，那么你就有多个 `root` 项目。
`wx.TR_NO_LINES`	如果设置了这个样式，那么树控件将不在兄弟项目间绘制连接线。这个样式将代替 `wx.TR_LINES_AT_ROOT` 。
`wx.TR_ROW_LINES`	树控件在行之间将绘制边距。

第三类样式显示在表15.3中，用于控制树控件的选择模式。

表**15.3** 树控件的选择模式

`wx.TR_EXTENDED`	可以选择多个不连续的项。不是对所有的系统有效。
`wx.TR_MULTIPLE`	可以选择一块且仅一块连续的项。
`wx.TR_SINGLE`	一次只能选择一个结点。这是默认模式。

表15.4显示了其它的一样可作用于树的显示的样式。

表**15.4** 树的其它显示样式

wx.TR_FULL_ROW_HIGHLIGHT	如果设置了这个样式，那么当被选择时，被选项的整个行将高亮显示。默认情况下，只是文本区高亮。 在 Windows 上，该样式只在也设置 wx.NO_LINES 时有效。
wx.TR_HAS_VARIABLE_ROW_HEIGHT	如果设置了这个样式，则行的高度将根据其中的图像和文本而不同。否则，所有的行将是同样的高度（取决于最高的行）。
wx.TR_HIDE_ROOT	如果设置了这个样式，则通过 AddRoot() 确定的 root 元素将不被显示。此时该结节的所有子项就如同它们是 root 一样显示。这个样式用于让一个树具有多个 root 元素的外观。

最后， wx.TR_DEFAULT_STYLE 让你的树显示出最接近当前操作系统本地控件的样式。在程序中，你可以使用 SetWindowStyle(styles) 来改变样式，其中参数 styles 是你想要的新样式。

树形控件有几个方法可以用以改变它的显示特性。在这些方法中，参数 item 是你想要改变的项的 wx.TreeItemId 。你可以使用方法 SetItemBackgroundColor(item, col) 来设置项目的背景色，其中参数 col 是一个 wx.Colour 或其它能够转换为颜色的东西。你可以使用 SetItemTextColour(item,col) 来改变文本的颜色。你可以使用 SetItemFont(item, font) 来设置项目的显示字体，其中参数 font 是一个 wx.Font 实例。如果你只想显示文本为粗体，你可以使用方法 SetItemBold(item, bold=True) ，其中参数 bold 是一个布尔值，它决定是否显示为粗体。上面的四个 set 方法都有对应的 *get* 方法，如下所示： GetItemBackgroundColor(item), GetItemTextColour(item), GetItemFont(i 和 IsBold(item) 。其中的 item 参数是一个 wx.TreeItemId 。

对树形控件的元素排序

对树形控件的元素排序的基本机制是方法 SortChildren(item) 。其中参数 item 是 wx.TreeItemId 的一个实例。该方法对此项目的子项目按显示字符串的字母的顺序进行排序。

对于树的排序，每个树项目都需要有已分派的数据，不管你是否使用默认排序。在默认情况中，所指派的数据是 None ，但是对于排序任务，在树控件中你还是需要显式地设置。

在15.1节，我们提及到了让你能够创建一个树项目及将该项目与一个任意数据对象相关联的方法。我们也告诉你不要使用这个机制。数据项目是一个`wx.TreeItemData`。在`wxPython`中在一预定义的快捷的方法，使你能够用以将一个`Python`对象与一个树项目关联起来。

快捷的`set`*方法是`SetItemPyData(item, obj)`。其中参数`item`是一个`wx.TreeItemId`，参数`obj`是一个任意的`Python`对象，`wxPython`在后台管理这个关联。当你想得到这个数据项时，你可以调用`GetItemPyData(item)`，它返回相关的`Python`对象。

注意：对于`wx.TreeItemData`有一个特别的构造函数：`wx.TreeItemData(obj)`，其中参数`obj`是一个`Python`对象。因此你可以使用`GetItemData(item)`和`SetItemData(item, obj)`方法来处理这个`Python`数据。这是`SetItemPyData()`方法的后台机制。这一信息在某些时候可能对你是有用的，但是大部分时候，你还是应该使用`SetItemPyData(item, obj)`和`GetItemPyData(item)`方法。

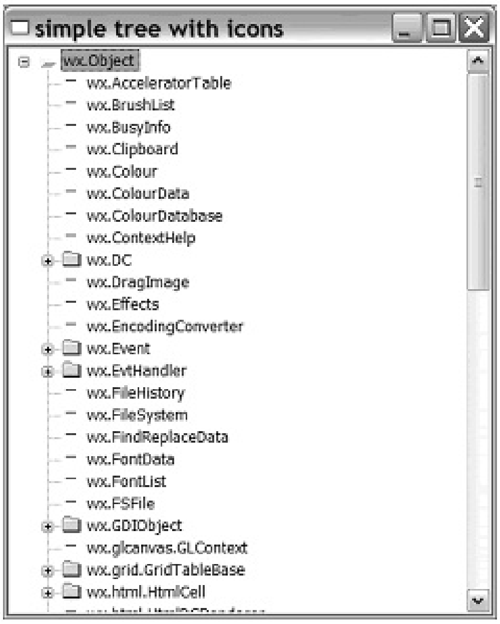
要使用关联的数据来排序你的树，你的树必须是一个`wx.TreeCtrl`的自定义的子类，并且你必须覆盖`OnCompareItems(item1, item2)`方法。其中参数`item1, item2`是要比较的两个项的`wx.TreeItemId`实例。如果`item1`应该排在`item2`的前面，则方法返回-1，如果`item1`应该排在`item2`的后面，则返回1，相等则返回0。该方法是在当树控件为了排序而比较计算每个项时自动被调用的。你可以在`OnCompareItems()`方法中做你想做的事情。尤其是，你可以如下调用`GetItemPyData()`方法：

```
def OnCompareItems(self, item1, item2);
    data1 = self.GetItemPyData(item1)
    data2 = self.GetItemPyData(item2)
    return cmp(data1, data2)
```

控制与每项相关的图像

用于树形控件的图像是由一个图像列表来维护的，这非常类似于列表控件中的图像维护。有关创建图像列表的细节，参见13章。一旦你创建了图像列表，你就可以使用`SetImageList(imageList)`或`AssignImageList(imageList)`方法把它分配给树控件。前者使得图像列表可以被其它控件共享，后者的图像列表所有权属于树控件。之后，你可能使用方法`GetImageList()`来得到该图像列表。图15.2显示了一个带有一些图像的树。

图15.2



例15.2是产生图15.2的代码。它使用了`ArtProvider`对象来提供图像。

例**15.2** 一个带有图标的树控件

```
#-*- encoding:UTF-8 -*-
import wx
import data
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None,
                title="simple tree with icons", size=(400,500))
        # 创建一个图像列表
        il = wx.ImageList(16,16)
        # 添加图像到列表
        self.fldridx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FOLDER,
                    wx.ART_OTHER, (16,16)))
        self.fldropenidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FILE_OPEN,
                    wx.ART_OTHER, (16,16)))
        self.fileidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_NORMAL_FILE,
                    wx.ART_OTHER, (16,16)))

        # 创建树
        self.tree = wx.TreeCtrl(self)
        # 给树分配图像列表
        self.tree.AssignImageList(il)
        root = self.tree.AddRoot("wx.Object")
        self.tree.SetItemImage(root, self.fldridx,
                               wx.TreeItemIcon_Normal)# 设置根的图
像
        self.tree.SetItemImage(root, self.fldropenidx,
                               wx.TreeItemIcon_Expanded)
```

```
        self.AddTreeNodes(root, data.tree)
        self.tree.Expand(root)

    def AddTreeNodes(self, parentItem, items):
        for item in items:
            if type(item) == str:
                newItem = self.tree.AppendItem(parentItem, item)
                self.tree.SetItemImage(newItem, self.fileidx,
                                       wx.TreeItemIcon_Normal)#
设置数据图像
            else:
                newItem = self.tree.AppendItem(parentItem, item[
0])
                self.tree.SetItemImage(newItem, self.fldridx,
                                       wx.TreeItemIcon_Normal)#
设置结点的图像
                self.tree.SetItemImage(newItem, self.fldropenidx
,
                                       wx.TreeItemIcon_Expanded)

                self.AddTreeNodes(newItem, item[1])

    def GetItemText(self, item):
        if item:
            return self.tree.GetItemText(item)
        else:
            return ""

app = wx.PySimpleApp(redirect=True)
frame = TestFrame()
frame.Show()
app.MainLoop()
```

如你所见，当你添加项目到列表中时，对于未选和选中状态，有两个不同的图像可用于分配给项目。和列表控件一样，你可以指定图像在图像列表中的索引。如果你想在项目被创建后得到所分配的图像，你可以使用方法 `GetItemImage(item, which=wx.TreeItemIcon_Normal)` 。其中参数 `item` 是项目的 `wx.TreeItemId` 。参数 `which` 控制你将得到了是哪个图像，默认值 `wx.TreeItemIcon_Normal` ，将得到该项目的未选状态的图像的索引。 `which` 的另一个值 `wx.TreeItemIcon_Selected` 的使用，将返回选中状态的图像， `wx.TreeItemIcon_Expanded` 和 `wxTreeItemIcon_SelectedExpanded` 返回当该树项目被展开时所使用的图像。注意，后者的两个图像不能使用添加方法来被设置——如果你想设置的话，你必须使用方法 `SetItemImage(item, image, which=wx.TreeItemIcon_Normal)` 来实现。其中参数 `item` 是 `wx.TreeItemId` 的实例，参数 `image` 是新图像的整数索引，参数 `which` 同 `get` *方法。

使用编程的方式访问树。

在15.1节中，我们谈到没有直接得到一个给定项目的子项目的 `Python` 列表的方法，至于一个特定子项目的索引就更不用说了。要实现这个，我们需要使用本节所说的遍历方法来访问树的结点。

要开始遍历树，我们首先要使用 `GetRootItem()` 来得到根元素。该方法返回树的根元素的 `wx.TreeItemId` 。之后，你就可以使用诸如 `GetItemText()` 或 `GetItemPyData()` 之类的方法来获取关于根元素的更多的信息。

一旦你得到了一个项目，你就可以通过迭代器来遍历子项目的列表。你可以使用方法 `GetFirstChild(item)` 来得到子树中的第一个孩子，该方法返回一个二元元组(`child, cookie)` 。参数 `item` 是第一个孩子的 `wx.TreeItemId` 。除了告诉你每一个孩子是什么外，该方法还初始化一个迭代对象，该对象使你能够遍历该树。 `cookie` 值只是一个标志，它使得树控件能够同时保持对多个迭代器的跟踪，而它们之间不会彼此干扰。

一旦你由 `GetFirstChild()` 得到了 `cookie` ，你就可以通过反复调用 `GetNextChild(item, cookie)` 来得到其余的孩子。这里的 `item` 父项的 `ID` ， `cookie` 是由 `GetFirstChild()` 或前一个 `GetNextChild()` 调用返回的。 `GetNextChild()` 方法也返回一个二元元组(`child, cookie)` 。如果此时没有下一个项目了，那么你就到达了孩子列表的末尾，系统将返回一个无效的孩子 `ID` 。你可以通过使用 `wx.TreeItemId.IsOk()` 或 `__nonzero__` 方法测试这种情况。下面的函数返回给定项目的所有孩子的文本的一个列表。

```
def getChildren(tree, parent):
    result = []
    item, cookie = tree.GetFirstChild(parent)
    while item:
        result.append(tree.GetItemText(item))
    item, cookie = tree.getNextChild(parent, cookie)
    return result
```

这个函数得到给定父项目的第一个孩子，然后将第一个孩子的文本添加到列表中，然后通过循环来得到每个子项目的文本并添加到列表中，直到得到一个无效的项，这时就返回 `result` 。

要得到父项目的最后一个孩子，你可以使用方法 `GetLastChild(item)` ，它返回列表中的最后的项目的 `wx.TreeItemId` 。由于这个方法不用于驱动迭代器来遍历整个孩子列表，所以它不需要 `cookie` 机制。如果你有这个最后的子项且你想得到它的父项，可以使用方法 `GetItemParent(item)` 来返回给定项的父项的 `ID` 。

你可以使用方法 `GetNextSibling(item)` 和 `GetPrevSibling(item)` 来在同级的项目间前后访问。这些方法均返回相关项的 `wx.TreeItemId` 。由这些方法同样不用于驱动迭代器，所以它们都不需要一个 `cookie` 。当你已经到达列表的两头

时，将没有下一项或前一项，那么这些方法将返回一个无效的项（例如：`item.IsOk() == False`）。

要确定一个项是否有孩子，使用方法`ItemHasChildren(item)`，该方法返回布尔值`True`或`False`。你可以使用方法`SetItemHasChildren(item, hasChildren=True)`来将一个项设置为有子项，如果这样，即使该项没有实际的子项，它也将显示得与有子项的一样。也就是说该项旁边会有一个扩展或折叠的按钮，以便展开或折叠。这通常被用于实现一个虚的树控件。这个技术将在15.7节中演示。

管理树中的选择

树形控件允许你通过程序的方式管理树中的被选项。基本的方法是`SelectItem(item,select=True)`。在单选树控件中，该方法将选择指定项`item`（`wx.TreeItemId`），同时自动撤消对先前选项的选择。如果参数`select`的取值是`False`，那么该方法将取消对参数`item`代表的项的选择。在一个可多选的树控件中，`SelectItem()`方法只改变`item`所代表的项的状态，而不改变树中其它项的选择状态。在可多选的树中，你也可以使用方法`ToggleItemSelection(item)`，它只切换参数`item`所代表项的选择状态。

对于取消选择还有三个快捷的方法。方法`Unselect()`取消单选模式树中的当前被选项的选择。在多选模式树中，使用`UnselectAll()`来取消所有的选择。如果你只想取消在一个多选树中的一个项的被选状态，可以使用方法`UnselectItem(item)`。

你也可以使用方法`IsSelected(item)`来查询一个项目的选择状态，该方法返回布尔值`True`或`False`。对于单选树，你可能使用方法`GetSelection()`来得到当前被选项目的`wx.TreeItemId`。对于多选树，可以使用方法`GetSelections()`来得到所有被选项的`wx.TreeItemId`的一个`Python`列表。

当树控件中发生选择变化的时候，有两个事件将被触发并可被捕获。第一个事件是`wx.EVT_TREE_SEL_CHANGING,`它在被选项实际改变之前发生。如果你要处理这个事件，你可以使用事件的`Veto()`方法来阻止选择的改变。在选择已经改变之后，事件`wx.EVT_TREE_SEL_CHANGED`被触发。这两个事件的类是`wx.TreeEvent`，这将在15.8节中作更完整的讨论。

控制项目的可见性

在树控件中有两种机制可以让你用编程的方式控制某项目的可见性。你可以使用方法`Collapse(item)`和`Expand(item)`指定给定的树项目是展开的或折叠的。这些方法改变树控件的显示，并且如果对一个没有子项的项目调用该方法将不起作用。这儿还有一个方便的函数：`CollapseAndReset(item)`，该方法折叠指定的项，并删除指定项的所有孩子。另外，方法`Toggle(item)`用于切换项目的展开和折叠状态。你可以使用方法`IsExpanded(item)`来查询项目的当前展开状态。

展开或折叠一个树项目触发两事件。在展开或折叠前，事件 `wx.EVT_TREE_ITEM_COLLAPSING` 或 `wx.EVT_TREE_ITEM_EXPANDING` 被触发。在你的处理方法中，你可以使用事件的 `Veto()` 方法来阻止展开或折叠。在展开或折叠发生后，事件 `EVT_TREE_ITEM_COLLAPSED` 或 `wx.EVT_TREE_ITEM_EXPANDED` 被触发。这四个事件都是类 `wx.TreeEvent` 的事件类型。

虚树

展开和折叠项目的一个令人感兴趣的的用法是创建一个虚树，新项目仅当父项展开时添加。例15.3显示这样一个样列。

例**15.3** 展开时动态添加新的项目

```
#-*- encoding:UTF-8 -*-
import wx
import data
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="virtual tree with i
cons", size=(400,500))
        il = wx.ImageList(16,16)
        self.fldridx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FOLDER, wx.ART_OTHER
, (16,16)))
        self.fldropenidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FILE_OPEN,   wx.ART_
OTHER, (16,16)))
        self.fileidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_NORMAL_FILE, wx.ART_
OTHER, (16,16)))
        self.tree = wx.TreeCtrl(self)
        self.tree.AssignImageList(il)
        root = self.tree.AddRoot("wx.Object")
        self.tree.SetItemImage(root, self.fldridx,
                               wx.TreeItemIcon_Normal)
        self.tree.SetItemImage(root, self.fldropenidx,
                               wx.TreeItemIcon_Expanded)

        # Instead of adding nodes for the whole tree, just attac
h some
        # data to the root node so that it can find and add its
child
        # nodes when it is expanded, and mark it as having child
ren so
        # it will be expandable.
        self.tree.SetItemPyData(root, data.tree)#创建一个根
        self.tree.SetItemHasChildren(root, True)

        # Bind some interesting events
# 绑定事件
        self.Bind(wx.EVT_TREE_ITEM_EXPANDED, self.OnItemExpanded
```

```
, self.tree)
        self.Bind(wx.EVT_TREE_ITEM_COLLAPSED, self.OnItemCollaps
ed, self.tree)
        self.Bind(wx.EVT_TREE_SEL_CHANGED, self.OnSelChanged, se
lf.tree)
        self.Bind(wx.EVT_TREE_ITEM_ACTIVATED, self.OnActivated,
self.tree)
        self.Bind(wx.EVT_TREE_ITEM_EXPANDING, self.OnItemExpandi
ng, self.tree)
        self.tree.Expand(root)

    def AddTreeNodes(self, parentItem):#给父项目添加结节
        """
 Add nodes for just the children of the parentItem
 """
        items = self.tree.GetItemPyData(parentItem)
        for item in items:
            if type(item) == str:
                # a leaf node
                newItem = self.tree.AppendItem(parentItem, item)
                self.tree.SetItemImage(newItem, self.fileidx,
                                       wx.TreeItemIcon_Normal)
            else:
                # this item has children
                newItem = self.tree.AppendItem(parentItem, item[
0])
                self.tree.SetItemImage(newItem, self.fldridx,
                                       wx.TreeItemIcon_Normal)
                self.tree.SetItemImage(newItem, self.fldropenidx
,
                                       wx.TreeItemIcon_Expanded)
                self.tree.SetItemPyData(newItem, item[1])
                self.tree.SetItemHasChildren(newItem, True)

    def GetItemText(self, item):
        if item:
            return self.tree.GetItemText(item)
        else:
            return ""

    def OnItemExpanded(self, evt):
        print "OnItemExpanded: ", self.GetItemText(evt.GetItem()
)
    def OnItemExpanding(self, evt):#当展开时创建结点
        # When the item is about to be expanded add the first le
vel of child nodes
        print "OnItemExpanding:", self.GetItemText(evt.GetItem()
)
        self.AddTreeNodes(evt.GetItem())

    def OnItemCollapsed(self, evt):
        print "OnItemCollapsed:", self.GetItemText(evt.GetItem()
)
```

```
        # And remove them when collapsed as we don't need them a
ny longer
        self.tree.DeleteChildren(evt.GetItem())#折叠时删除结点
    def OnSelChanged(self, evt):
        print "OnSelChanged: ", self.GetItemText(evt.GetItem())
    def OnActivated(self, evt):
        print "OnActivated: ", self.GetItemText(evt.GetItem())
app = wx.PySimpleApp(redirect=True)
frame = TestFrame()
frame.Show()
app.MainLoop()
```

这个机制可以被扩展来从外部源读取数据以查看。这个机制除了可以被用来建造一个文件树外，我们会提及数据库中的数据的可能性，以便对文件的结构不感趣的你不用全面研究文件结构。

控制可见性

有大量的方法使你能够管理那些项目是可见的。一个对象的不可见，可能是因为它没处在含有滚动条的框中的可见区域或是因为它处在折叠项中。你可以使用 `IsVisible(item)` 方法来确定项目是否可见，该方法在项目是可见时返回 `True` ，不可见返回 `False` 。你可以通过使用方法 `EnsureVisible(item)` 迫使指定的项变成可见的。如果需要的话，该方法将通过展开该项的父项（以及父项的父项，以此类推）来迫使指定项变成可见的，然后滚动该树，以使指定项处于控件的可见部分。如果你只需要滚动，可以使用 `ScrollTo(item)` 方法来完成。

遍历树中的可见部分，首先要使用的方法是 `GetFirstVisibleItem()` 。该方法返回显示的可见部分中的最顶端的项目的 `wx.TreeItemId` 。然后通过使用 `GetNextVisible(item)` 方法来遍历，该方法的 `item` 参数来自 `GetFirstVisibleItem()` 和 `GetNextVisible()` 的返回值。如果移动的方向向上的话，使用方法 `GetPreviousVisible(item)` 。如果参数 `item` 不可见的话，返回值是一个无效的项。

还有几个别的方法可用于项目的显示。树控件有一个属性，该属性用于设置缩进。可以通过方法 `GetIndent()` 和方法 `SetIndent(indent)` 来得到和设置该属性的当前值。其中 `indent` 参数是缩进的像素值（整数）。

要得到关于指定点的树项目的信息，使用方法 `HitTest(point)` ，其中 `point` 是树控件中的相关位置的一个 `wx.Point` 。方法的返回值是一个(`item, flags)` 元组，其中的 `item` 是相关位置的项的 `wx.TreeItemId` 或 `None` 值。如果指定位置没有项目，那么一个无效的项目将返回。 `flags` 部分是一个位掩码，它给出了相关的信息。表15.5包含了 `flags` 的一个完整列表。

还有两个方法，它们让你可以处理屏幕上项目的实际的边界。方法 `GetBoundingRect(item, textOnly=False)` 返回一个 `wx.Rect` 实例，该实例对应于屏幕上文本项的矩形边界区域。其中参数 `item` 是项目的 `wx.TreeItemId` 。如果参数 `textOnly` 为 `True` ，那么该矩形仅包括项目的

显示文本所覆盖的区域。如果为 `False` ，那么该矩形也包括图像区域。在这两种情况中，矩形都包括从树控件的边缘到内嵌的显示项间的空白区域。如果 `item` 代表的项目当前是不可见的，那么两种方法都返回 `None` 。

表**15.5**

`wx.TREE_HITTEST_ABOVE`	该位置在树的客户区的上面，不是任何项目的一部分。
`wx.TREE_HITTEST_BELOW`	该位置在树的客户区的下面，不是任何项目的一部分。
`wx.TREE_HITTEST_NOWhere`	该位置处在树的客户区中，不是任何项目的一部分。
`wx.TREE_HITTEST_ONITEMBUTTON`	该位置位于展开/折叠图标按钮上，是项目的一部分。
`wx.TREE_HITTEST_ONITEMICON`	该位置位于项目的图像部分上。
`wx.TREE_HITTEST_ONITEMINDENT`	该位置位于项目的显示文本的左边缩进区域中。
`wx.TREE_HITTEST_ONITEMLABEL`	该位置位于项目的显示文本中。
`wx.TREE_HITTEST_ONITEMRIGHT`	该位置是项目的显示文本的右边。
`wx.TREE_HITTEST_ONITEMSTATEICON`	该位置是在项目的状态图标中。
`wx.TREE_HITTEST_TOLEFT`	该位置在树的客户区的左面，不是任何项目的一部分。
`wx.TREE_HITTEST_TORIGHT`	该位置在树的客户区的右面，不是任何项目的一部分。

使树控件可编辑

树控件可以被设置为允许用户编辑树项目的显示文本。这通过在创建树控件时使用样式标记 `wx.TR_EDIT_LABELS` 来实现。使用了该样式标记后，树控件的行为就类似于可编辑的列表控件了。编辑一个树项目会给出一个文本控件来让用户编辑文本。按下 `esc` 则取消编辑。按下回车键或在文本控件外敲击将确认编辑。

你可以在程序中使用 `EditLabel(item)` 方法来启动对特定项的编辑。参数 `item` 是你想要编辑的项的 `wx.TreeItemId` 。要终止编辑，可以使用方法 `EndEditLabel(cancelEdit)` 。由于一次只能有一个活动编辑项，所以这里不需要指定项目的 `ID` 。参数 `cancelEdit` 是一个布尔值。如果为 `True` ，取消编辑，如果为 `False` ，则不取消。如果因为某种原因，你需要访问当前所用的文本

编辑控件，你可以调用方法 `GetEditControl()` ，该方法返回用于当前编辑的 `wx.TextCtrl` 实例，如果当前没有编辑，则返回 `None` 。当前该方法只工作于 `Windows` 系统下。

当一个编辑会话开始时（通过用户选择或调用 `EditLabel()` 方法）， `wx.EVT_TREE_BEGIN_LABEL_EDIT` 类型的 `wx.TreeEvent` 事件被触发。如果使用 `Veto()` 方法否决了该事件的话，那么编辑将不会开始。当会话结束时（通过用户的敲击或调用 `EndEditLabel()` 方法），一个 `wx.EVT_TREE_END_LABEL_EDIT` 类型的事件被触发。这个事件也可以被否决，这样的话，编辑就被取消了，项目不会被改变。

响应树控件的其它的用户事件

在这一节，我们将讨论 `wx.TreeEvent` 类的属性。表15.6列出了这些属性。

表15.6 `wx.TreeEvent` 的属性

`GetKeyCode()`	返回所按键的整数按键码。只对 `wx.EVT_TREE_KEY_` 果任一修饰键（ `CTRL,SHIFT,and ALT` 之类的）也 你。
`GetItem()`	返回与事件相关的项的 `wx.TreeItemId` 。
`GetKeyEvent()`	只对 `wx.EVT_TREE_KEY_DOWN` 事件有效。返回 `wx.` 被用来告知你在该事件期间，是否有修饰键被按下。
`GetLabel()`	返回项目的当前文本标签。只对 `wx.EVT_TREE_BEGIN_LABEL_EDIT` 和 `wx.EVT_T` 效。
`GetPoint()`	返回与该事件相关的鼠标位置的一个 `wx.Point` 。
`IsEditCancelled()`	只对 `wx.EVT_TREE_END_LABEL_EDIT` 有效。如果用 编辑则返回 `True` ，否则返回 `False` 。
`SetToolTip(tooltip)`	只对 `wx.EVT_TREE_ITEM_GETTOOLTIP` 事件有效。 的提示。该属性只作在 `Windows` 系统上。

表15.7列出了几个不适合在表15.6中列出的 `wx.TreeEvent` 的事件类型，它们有时也是用的。

表15.7 树控件的另外的几个事件

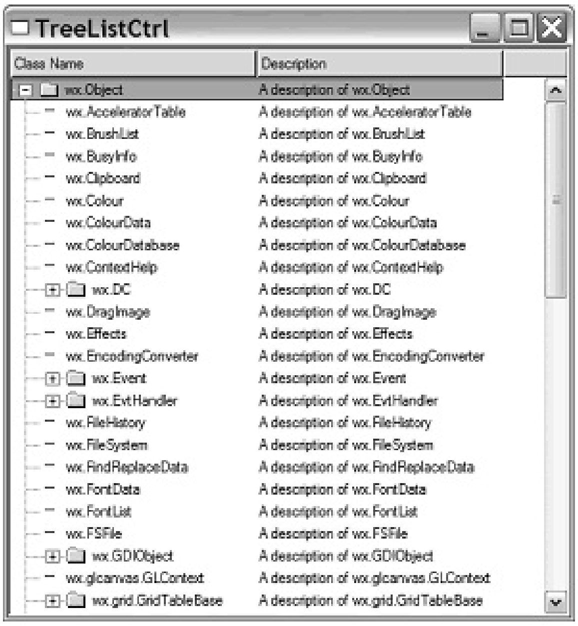
`wx.EVT_TREE_BEGIN_DRAG`	当用户通过按下鼠标左键来拖动树中的一个项目时，触发该事件。要让拖动时能够做些事情，该事件的处理函数必须显式地调用事件的 `Allow()` 方法。
`wx.EVT_TREE_BEGIN_RDRAG`	当用户通过按下鼠标右键来拖动树中的一个项目时，触发该事件。要让拖动时能够做些事情，该事件的处理函数必须显式地调用事件的 `Allow()` 方法。
`wx.EVT_TREE_ITEM_ACTIVATED`	当一个项目通过双击被激活时，触发该事件。
`wx.EVT_TREE_ITEM_GETTOOLTIP`	当鼠标停留在树中的一个项目上时，触发该事件。该事件可用来为项目设置特定的提示。只需要在事件对像中简单地设置标签参数，其它的将由系统来完成。
`wx.EVT_TREE_KEY_DOWN`	在树控件获得焦点的情况下，当一个键被按下时触发该事件。

上面这些就你需要了解的有关树控件的属性。下面，我们将通过一个有用的另一种形式的树控件来结束本章。

使用树列表控件

除了 `wx.TreeCtrl` ， `wxPython` 也提供了 `wx.gizmos.TreeListCtrl` ，它是树控件和报告模式的列表控件的组合。除了本章中所讨论的 `wx.TreeCtrl` 的特性外， `TreeListCtrl` 能够显示与每行相关的数据的附加列。图15.3显示了树列表控件的样子。

图**15.3**



例15.4是产生上图的代码

例**15.4** 使用树列表控件

```
import wx
import wx.gizmos
import data
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="TreeListCtrl", size
=(400,500))
        # Create an image list
        il = wx.ImageList(16,16)
        # Get some standard images from the art provider and add
 them
        # to the image list
        self.fldridx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FOLDER, wx.ART_OTHER
, (16,16)))
        self.fldropenidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_FILE_OPEN,   wx.ART_
OTHER, (16,16)))
        self.fileidx = il.Add(
            wx.ArtProvider.GetBitmap(wx.ART_NORMAL_FILE, wx.ART_
OTHER, (16,16)))

        # Create the tree
# 创建树列表控件
        self.tree = wx.gizmos.TreeListCtrl(self, style =
                                           wx.TR_DEFAULT_STYLE
                                           | wx.TR_FULL_ROW_HIGH
LIGHT)
        # Give it the image list
        self.tree.AssignImageList(il)
```

```
        # create some columns
#创建一些列
        self.tree.AddColumn("Class Name")
        self.tree.AddColumn("Description")
        self.tree.SetMainColumn(0) # the one with the tree in it
...
        self.tree.SetColumnWidth(0, 200)
        self.tree.SetColumnWidth(1, 200)
        # Add a root node and assign it some images
        root = self.tree.AddRoot("wx.Object")
        self.tree.SetItemText(root, "A description of wx.Object"
, 1)#给列添加文本
        self.tree.SetItemImage(root, self.fldridx,
                               wx.TreeItemIcon_Normal)
        self.tree.SetItemImage(root, self.fldropenidx,
                               wx.TreeItemIcon_Expanded)

        # Add nodes from our data set
        self.AddTreeNodes(root, data.tree)
        # Bind some interesting events
        self.Bind(wx.EVT_TREE_ITEM_EXPANDED, self.OnItemExpanded
, self.tree)
        self.Bind(wx.EVT_TREE_ITEM_COLLAPSED, self.OnItemCollaps
ed, self.tree)
        self.Bind(wx.EVT_TREE_SEL_CHANGED, self.OnSelChanged, se
lf.tree)
        self.Bind(wx.EVT_TREE_ITEM_ACTIVATED, self.OnActivated,
self.tree)
        # Expand the first level
        self.tree.Expand(root)

    def AddTreeNodes(self, parentItem, items):
        """
 Recursively traverses the data structure, adding tree nodes to
 match it.
 """
        for item in items:
            if type(item) == str:
                newItem = self.tree.AppendItem(parentItem, item)
                self.tree.SetItemText(newItem, "A description of
 %s" % item, 1)#给列添加文本
                self.tree.SetItemImage(newItem, self.fileidx,
                                       wx.TreeItemIcon_Normal)
            else:
                newItem = self.tree.AppendItem(parentItem, item[
0])
                self.tree.SetItemText(newItem, "A description of
 %s" % item[0], 1)
                self.tree.SetItemImage(newItem, self.fldridx,
                                       wx.TreeItemIcon_Normal)
                self.tree.SetItemImage(newItem, self.fldropenidx
,
                                       wx.TreeItemIcon_Expanded)
```

```
                self.AddTreeNodes(newItem, item[1])

    def GetItemText(self, item):
        if item:
            return self.tree.GetItemText(item)
        else:
            return ""

    def OnItemExpanded(self, evt):
        print "OnItemExpanded: ", self.GetItemText(evt.GetItem()
)

    def OnItemCollapsed(self, evt):
        print "OnItemCollapsed:", self.GetItemText(evt.GetItem()
)
    def OnSelChanged(self, evt):
        print "OnSelChanged: ", self.GetItemText(evt.GetItem())
    def OnActivated(self, evt):
        print "OnActivated: ", self.GetItemText(evt.GetItem())
app = wx.PySimpleApp(redirect=True)
frame = TestFrame()
frame.Show()
app.MainLoop()
```

树列表控件的很多方法和列表控件的相似，所以我们就再也没有列出并说明了。

本章小结

1、树控件提供给你了一个如文件树或 `XML` 文档样的嵌套紧凑的外观。树控件是类 `wx.TreeCtrl` 的实例。有时，人你会想子类化 `wx.TreeCtrl` ，尤其是如果你需要实现自定义的排序的时候。

2、要向树中添加项目，首先要用方法 `AddRoot(text, image=` -1, `selImage=` -1, `data=None)` 。该函数的返回值是一个代表树的 `root` 项目的 `wx.TreeItemId` 。树控件使用 `wx.TreeItemId` 作为它自己的标识符类型，而非和其它控件一样使用整数的 `ID` 。一旦你得到了 `root` 项，你就可以使用方法 `AppendItem(parent, text, image=` -1, `selImage=` -1, `data=None)` 来添加子项目，参数 `parent` 是父项目的 `ID` 。该方法返回新项目的 `wx.TreeItemId` 。另外还有一些用来将新的项目添加在不同位置的方法。方法 `Delete(item)` 从树中移除一个项目，方法 `DeleteChildren(item)` 移除指定项的所有子项目。

3、树控件有一些用来改变树的显示外观的样式。一套是用来控制展开或折叠项目的按钮类型的。另一套是用来控制项目间的连接线的。第三套是用于控制树是单选还是多选的。你也可以使用样式通过隐藏树的实际的 `root` ，来模拟一个有着多个 `root` 项的树。

4、默认情况下，树的项目通常是按照显示文本的字母顺序排序的。但是要使这能够实现，你必须给每项分配数据。实现这个的最容易的方法是使用 `SetItemPyData(item, obj)` ，它给项目分配一个任意的 `Python` 对象。如果你想使用该数据去写一个自定义的排序函数，你必须继承 `wx.TreeCtrl` 类并覆盖方法 `OnCompareItems(item1, item2)` ，其中的参数 `item1` 和 `item2` 是要比较的项的 `ID` 。

5、树控件使用一个图像列表来管理图像，类似于列表控件管理图像的方法。你可以使用 `SetImageList(imageList)` 或 `AssignImageList(imageList)` 来分配一个图像列表给树控件。然后，当新的项目被添加到列表时，你可以将它们与图像列表中的特定的索引联系起来。

6、没有特定的函数可以让你得到一个父项的子项目列表。替而代之的是，你需要去遍历子项目列表，这通过使用方法 `GetFirstChild(item)` 作为开始。

7、你可以使用方法 `SelectItem(item, select=True)` 来管理树的项目的选择。在一个多选树中，你可以使用 `ToggleItemSelection(item)` 来改变给定项的状态。你可以使用 `IsSelected(item)` 来查询一个项目的状态。你也可以使用 `Expand(item)` 或 `Collapse(item)` 展开或折叠一个项，或使用 `Toggle(item)` 来切换它的状态。

8、样式 `wx.TR_EDIT_LABELS` 使得树控件可为用户所编辑。一个可编辑的列表中，用户可以选择一个项，并键入一个新的标签。按下 `esc` 来取消编辑而不对项目作任何改变。你也可以通过 `wx.EVT_TREE_END_LABEL_EDIT` 事件来否决编辑。类 `wx.TreeEvent` 提供了允许访问当前被处理的项目的显示文本的属性。

第十六章 在应用程序中加入HTML

本章内容：

- 在 wxPython 窗口中显示 HTML
- 处理和打印 HTML 窗口
- 使用 HTML 分析器(parser)
- 支持新的标记和其它的文件格式
- 在 HTML 中使用控件

HTML 最初是打算被作为超文本系统使用的一个简单的语义标记来使用的。迄今为止， HTML 已经变得更加的复杂和被广泛使用。 HTML 文档标记已经被证明在网页浏览器之外也是有用的。目前 HTML 文档标记通常被用于文本标记（如在文本控件中），或用于管理一系列的超链接页面（帮助系统中）。在 wxPython 中，有许多专用于处理你的 HTML 需求的特性。你可以在一个窗口中显示简单的 HTML ，并用超链接创建你自己的帮助页面，如果你需要的话，甚至你还可以嵌入一个功能更全的浏览器。

下一节内容提示：如何在 wxPython 窗口中显示 HTML ？

显示HTML

在 wxPython 中，你对 HTML 能做的最重要的事情就是将它显示在一个窗口中。下面的两节，我们将讨论 HTML 窗口对象，以及给你展示如何对本地的文本或远程的 URL 使用它。

如何在一个wxPython窗口中显示HTML？

正如我们在第六章中讨论的，对于使用样式文本或简单的网格来快速地描述文本的布局， wxPython 中的 HTML 是一个有用的机制。 wxPython 的 wx.html.HtmlWindow 类就是用于此目的的。图16.1显示了一个例子。

图**16.1**



Figure 16.1 A very simple HtmlWindow

例16.1显示了用于产生图16.1的代码。

例**16.1** 显示简单地`HtmlWindow`

`import wx`

```
import wx.html

class MyHtmlFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title)
        html = wx.html.HtmlWindow(self)
        if "gtk2" in wx.PlatformInfo:
            html.SetStandardFonts()

        html.SetPage(
            "Here is some  b formatted /b   i  u text /u  /i "
            "loaded from a  font color=\"red\" string /font .")

app = wx.PySimpleApp()
frm = MyHtmlFrame(None, "Simple HTML")
frm.Show()
app.MainLoop()
```

`wx.html.HtmlWindow`的构造函数基本上是与`wx.ScrolledWindow`相同的，如下所示：

```
wx.html.HtmlWindow(parent, id=-1, pos=wx.DefaultPosition,
size=wx.DefaultSize, style=wx.html.HW_SCROLLBAR_AUTO,
name="htmlWindow")
```

上面的这些参数现在看着应该比熟悉。这最重要的不同点是默认样式`wx.html.HW_SCROLLBAR_AUTO`，它将告诉`HTML`窗口在需要的时候自动增加滚动条。与之相反的样式是`wx.html.HW_SCROLLBAR_NEVER`，使用该样式将不会显示滚动条。还有一个`HTML`窗口样式是`wx.html.HW_NO_SelectION`，它使得用户不能选择窗口中的文本。

当在`HTML`窗口中写要显示的`HTML`时，记住所写的`HTML`要是简单的。因为`wx.html.HtmlWindow`控件仅设计用于简单样式文本显示，而非用于全功能的多媒体超文本系统。它只支持最基本的文本标记，更高级的特性如层叠样式表(`css`)和`JavaScript`不被支持。表16.1包含了官方支持的`HTML`标记。通常，这里的标记和它的属性的行为和`web`浏览器中的一样，但是由于它不是一个完全成熟的浏览器，所以有时会出现一些奇怪行为的情况。表16.1中列出了后跟有属性的标记。

表**16.1** 用于`HTML`窗口控件的有效的标记

文档结构标记	`a href na`
`link text meta content http-equiv title`	
文本结构标记	`br div al`
文本显示标记	`address b`
`code em font color face size h1 h2 h3 h4 h5 h6`	
`i kbd pre samp small strike string tt u`	
列表标记	`dd dl dt`
图像和地图标记	`area coor`
`height src width usemap map name`	
表格标记	`table ali`
`cellspacing valign width td align bgcolor colspan`	
`rowspan valign width nowrap th align bgcolor colspan`	
`valign width rowspan tr align bgcolor valign`	

`HTML` 窗口使用 `wx.Image` 来装载和显示图像，所以它可以支持所有 `wx.Image` 支持的图像文件格式。

如何显示来自一个文件或URL的HTML？

一旦你创建了一个 `HTML` 窗口，接下来就是在这个窗口中显示 `HTML` 文本。下面的四个方法用于在窗口中得到 `HTML` 文本。

- `SetPage(source)`
- `AppendToPage(source)`
- `LoadFile(filename)`
- `LoadPage(location)`

其中最直接的方法是 `SetPage(source)` ，参数 `source` 是一个字符串，它包含你想显示在窗口中的 `HTML` 资源。

你可以使用方法 `AppendToPage(source)` 添加 `HTML` 到窗口中的文本的后面。至于 `SetPage()` 和 `AppendToPage()` 方法，其中的参数 `source` 被假设是 `HTML` ，这意味着，如果你传递的是纯文本，那么其中的间距将被忽略，以符合 `HTML` 标准。

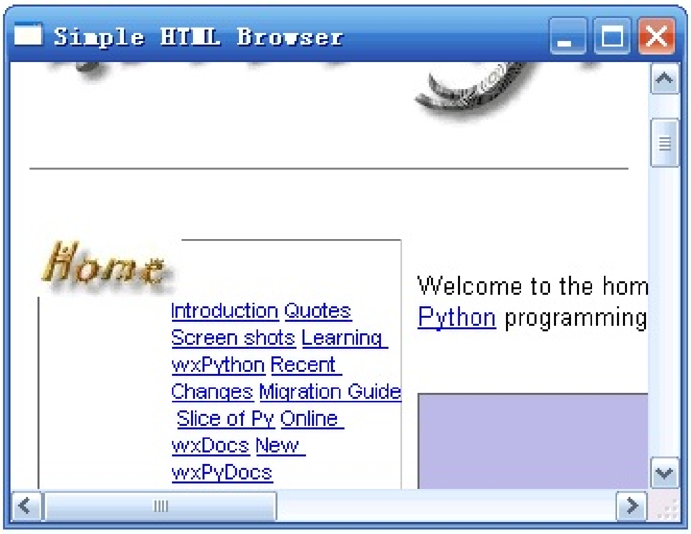
如果你想让你的窗口在浏览外部的资源时更像一个浏览器，那么你有两种方法。方法 `LoadFile(filename)` 读取本地文件的内容并将它们显示在窗口中。在这种情况中，窗口利用 `MIME` 文件类型来装载一个图像文件或一个 `HTML` 文件。如果它

不能确定文件是何种类型，那么它将以纯文本的方式装载该文件。如果被装载的文档包含有相关图像或其它文档的链接，那么被用于解析那些链接的位置是原文件的位置。

当然，一个实际的浏览器不会只局限于本地文件。你可以使用方法 `LoadPage(location)` 来装载一个远程的 `URL` ，其中参数 `location` 是一个 `URL` ，但是对于本地文件，它是一个路径名。 `MIME` 类型的 `URL` 被用来决定页面如何被装载。本章的稍后部分，我们将讨论如何增加对新文件类型的支持。

图16.2显示了被装载入 `HTML` 窗口中的一个页面。

图**16.2**



例16.2显示了产生图16.2的代码

例**16.2** 从一个 **`web`** 页装载 **`HTML`** 窗口的内容

```
import wx
import wx.html

class MyHtmlFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title, size=(600,400
))
        html = wx.html.HtmlWindow(self)
        if "gtk2" in wx.PlatformInfo:
            html.SetStandardFonts()

        wx.CallAfter(
            html.LoadPage, "http://www.wxpython.org")

app = wx.PySimpleApp()
frm = MyHtmlFrame(None, "Simple HTML Browser")
frm.Show()
app.MainLoop()
```

例16.2中关键的地方是方法 `LoadPage()` 。拥有更完整特性的浏览器窗口还应有显示 `URL` 的文本框，并在当用户键入一个新的 `URL` 后，可以改变窗口中的内容。

管理HTML窗口

一旦你有了一个 `HTML` 窗口，你就可以通过不同的方法来管理它。你可以根据用户的输入来触发相应的动作，处理窗口的内容，自动显示有关窗口的信息和打印页面等。在随后的几节中，我们将讨论如何实现这些。

如何响应用户在一个链接上的敲击？

`wx.html.HtmlWindow` 的用处不只限于显示。还可以用于响应用户的输入。在这种情况下，你不需要定义你自己的处理器，你可以在你的 `wx.html.HtmlWindow` 的子类中覆盖一些处理函数。

表16.2说明了已定义的处理函数。 `wx.html.HtmlWindow` 类没有使用事件系统定义事件，所以你必须使用这些重载的成员函数来处理相关的事件，而非绑定事件类型。

另外，如果你想让一个 `HTML` 窗口响应用户的输入，你必须创建你自己的子类并覆盖这些方法。

表16.2 `wx.html.HtmlWindow` 的事件处理函数

`OnCellClicked(cell, x, y, event)`	当用户在 `HTML` 文档中敲击时调用。 表所显示的文档的一部分，诸如文本 析器创建，这将在本章后部分讨论。 数 `event` 是相关的鼠标敲击事件。 单地委托给 `OnLinkClicked()` ，否
`OnCellMouseHover(cell, x, y)`	当鼠标经过一个 `HTML` 单元时调用。
`OnLinkClicked(link)`	当用户在一个超链接上敲击时调用。 方法通常用于使用 `HtmlWindow` 来 行为以便用户通过敲击其中的主页来 览器。
`OnOpeningURL(type, url)`	当用户请求打开一个 `URL` 时调用， 是 `wx.html.HTML_URL_PAGE, wx.h` 该方法返回下列值之一—— `wx.htm` 载入资源;或用于 `URL` 重定向的一个 默认版总是返回 `wx.html.HTML_OPE`
`OnSetTitle(title)`	当 `HTML` 源文件中有 `title` 标记时

如何使用编程的方式改变一个HTML窗口？

当你正显示一个 HTML 页时，你还可以改变你的窗口像浏览器样去显示其它的内容，如一另一个 Web 页，或帮助文件或其它类型的数据，以响应用户的需要。

有两个方法来当 HTML 窗口在运行时，访问和改变 HTML 窗口中的信息。首先，你可以使用 GetOpenedPage() 方法来得到当前打开的页面的 URL 。该方法只在当前页是被 LoadPage() 方法装载的才工作。如果是这样的，那么方法的返回值是当前页的 URL 。否则，或当前没有打开的页面，该方法返回一个空字符串。另一个相关的方法是 GetOpenedAnchor() ，它返回当前打开页面中的锚点（ anchor ）。如果页面不是被 LoadPage() 打开的，你将得到一个空的字符串。

要得到当前页的 HTML 标题，可以使用方法 GetOpenedPageTitle() ，这将返回当前页的 title 标记中的值。如果当前页没有一个 title 标记，你将得到一个空的字符串。

这儿有几个关于改变窗口中文本的选择的方法。方法 SelectAll() 选择当前打开的页面中的所有文本。你可以使用 SelectLine(pos) 或 SelectWord(pos) 做更有针对性的选择。其中 pos 是鼠标的位置 wx.Point ，这两个方法分别选择一行或一个词。要取得当前选择中的纯文本内容，可以使用方法 SelectionToText() ，而方法 ToText() 返回整个文档的纯文本内容。

wx.html.HtmlWindow 维护着历史页面的一个列表。使用下表16.3中的方法，可以如通常的浏览器一样浏览这个历史列表。

表16.3

HistoryBack()	装载历史列表中的前一项。如果不存在则返回 False 。
HistoryCanBack()	如果历史列表中存在前一项，则返回 True ，否则返回 False 。
HistoryCanForward()	如果历史列表中存在下一项，则返回 True ，否则返回 False 。
HistoryClear()	清空历史列表。
HistoryForward()	装载历史列表中的下一项。如果不存在则返回 False 。

要改变正在使用的字体，可以使用方法 SetFonts(normal_face, fixed_face, sizes=None) 。参数 normal_face 是你想用在窗口显示中的字体的名字字符串。如果 normal_face 是一个空字符串，则使用系统默认字体。参数 fixed_face 指定固定宽度的文本，类似于 pre 标记的作用。如果指定了 fixed_face 参数，那么参数 sizes 则应是一个代表字体的绝对尺寸的包含7个整数的列表，它对应于 HTML 逻辑字体尺寸（如 font 标记所使用的）－24之间。如果该参数没有指定或是 None ，则使用默认的。关于默认常量 wx.html.HTML_FONT_SIZE_n ，n位于1~7之间。这些默认常量指定了对应于 HTML 逻辑字体尺寸所使用的默认字

体。准确的值可能因不同的底层系统而不同。要选择一套基于用户的系统的字体和尺寸，可以调用 `SetStandardFonts()` 。这在 `GTK2` 下运行 `wxPython` 时是特别有用的，它能够提供一套更好的字体。

如果由于某种原因，你需要改变窗口中文本边缘与窗口边缘之间的间隔的话， `HTML` 窗口定义了 `SetBorders(b)` 方法。参数b是间隔的像素宽度（整数值）。

如何在窗口的标题栏中显示页面的标题？

在你的 `web` 浏览器中，你可能也注意到了一件事，那就是浏览器中不光只有显示窗口，还有标题栏和状态栏。通常，标题栏显示打开页面的标题，状态栏在鼠标位于链接上时显示链接信息。在 `wxPython` 中有两个便捷的方法来实现这些。图16.3对此作了展示。窗口显示的标题是基于 `web` 页面的标题的，状态栏文本也来自 `Html` 窗口。

例16.3是产生图16.3的代码。

图**16.3** 带有状态栏和标题栏的 **`HTML`** 窗口



例**16.3** 从一个 **`web`** 页载入 **`HTMLWindow`** 的内容

```
#-*- encoding:UTF-8 -*-
import wx
import wx.html

class MyHtmlFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title, size=(600,400
))
        self.CreateStatusBar()

        html = wx.html.HtmlWindow(self)
        if "gtk2" in wx.PlatformInfo:
            html.SetStandardFonts()
        html.SetRelatedFrame(self, self.GetTitle() + " -- %s") #
关联HTML到框架
        html.SetRelatedStatusBar(0) #关联HTML到状态栏

        wx.CallAfter(
            html.LoadPage, "http://www.wxpython.org")

app = wx.PySimpleApp()
frm = MyHtmlFrame(None, "Simple HTML Browser")
frm.Show()
app.MainLoop()
```

要设置标题栏的关联，使用方法 `SetRelatedFrame(frame, format)` 。参数 `frame` 你想显示页面标题的框架。参数 `format` 是你想在框架的标题栏中显示的字符串。通常的格式是这样：“ `My wxPython Browser: %s`”。:%s前面的字符串可以是你想要的任何字符串，%s将会被 `HTML` 页面的标题所取代。在窗口中，一个页面被载入时，框架的标题自动被新的页面的信息取代。

要设置状态栏，使用方法 `SetRelatedStatusBar(bar)` 。该方法必须在 `SetRelatedFrame()` 之后调用。参数 `bar` 是状态栏中用于显示状态信息的位置。通常它是0，但是如果状态栏中存在多个显示区域，那么 `bar` 可以有其它的值。如果 `bar` 的取值为－1,那么不显示任何信息。一旦与状态栏的关联被创建，那么当鼠标移动到显示的页面的链接上时，相关链接的 `URL` 将显示在状态栏中。

如何打印一个HTML页面？

一旦 `HTML` 被显示在屏幕上，接下来可能做的事就是打印该 `HTML` 。类 `wx.html.HtmlEasyPrinting` 就是用于此目的的。你可以使用下面的构造函数来创建 `wx.html.HtmlEasyPrinting` 的一个实例：

```
wx.html.HtmlEasyPrinting(name="Printing", parentWindow=None)
```

参数 `name` 只是一个用于显示在打印对话框中的字符串。参数 `parentWindow` 如果被指定了，那么 `parentWindow` 就是这些打印对话框的父窗口。如果 `parentWindow` 为 `None` ，那么对话框为顶级对话框。你只应该创建 `wx.html.HtmlEasyPrinting` 的一个实例。尽管 `wxPython` 系统没有强制要这样做，但是该类是被设计为独自存的。

使用 **`wx.html.HtmlEasyPrinting`** 的实例

从该类的名字可以看出，它应该是容易使用的。首先，通过使用 `PrinterSetup()` 和 `PageSetup()` 方法，你能够给用户显示用于打印设置的对话框。调用这些方法将导致相应的对话框显示给用户。实例将存储用户所做的设置，以备后用。如果你想访问这些设置数据，以用于你自己特定的处理，你可以使用方法 `GetPrintData()` 和 `GetPageSetupData()` 。 `GetPrintData()` 方法返回一个 `wx.PrintData` 对象， `GetPageSetupData()` 方法返回一 `wx.PageSetupDialogData` 对象，我们将在第17章中更详细地讨论。

设置字体

你可以使用方法 `SetFonts(normal_face, fixed_face, sizes)` 来设置打印所使用的字体。这个方法的行为同用于 `HTML` 窗口的 `SetFonts()` 相同（在打印对象中的设置不会影响到 `HTML` 窗口中的设置）。你可以使用方法 `SetHeader(header, pg)` 和 `SetFooter(footer, pg)` 来页眉和页脚。参数 `header` 和 `footer` 是要显示的字符串。字符串中你可以使用点位符 @ `PAGENUM` @，占位符在执行时被打印的页号替代。你也可以使用 @ `PAGENUM` @占位符，它是打印的页面总数。参数 `pg` 的取值可以是这三个： `wx.PAGE_ALL` 、 `wx.PAGE_EVEN` 或 `wx.PAGE_ODD` 。它控制页眉和页脚显示在哪个页上。通过对不同的 `pg` 参数多次调用该方法，可以为奇数页和偶数页设置单独的页眉和页脚。

输出预览

如果在打印前，你想预览一下输出的结果，你可以使用 `PreviewFile(htmlfile)` 方法。在这种情况下，参数 `htmlfile` 是你本地的包含 `HTML` 的文件的文件名。另一是 `PreviewText(htmlText, basepath="")`。参数 `htmlText` 是你实际想打印的 `HTML` 。 `basepath` 文件的路径或 `URL` 。如预览成功，这两个方法均返回 `True` ，否则返回 `False` 。如果出现了错误，那么全局方法 `wx.Printer.GetLastError()` 将得到更多的错误信息。关于该方法的更详细的信息将在第17章中讨论。

打印

现在你可能想知道如何简单地打印一个 `HTML` 页面。方法就是 `PrintFile(htmlfile)` 和 `PrintText(htmlText, basepath)` 。其中的参数同预览方法。所不同的是，这两个方法使用对话框中的设置直接让打印机打印。打印成功，则返回 `True` 。

拓展HTML窗口

在这一节，我们将给你展示如何处理 `HTML` 窗口中的 `HTML` 标记，如何创造你自己的标记，如何在 `HTML` 中嵌入 `wxPython` 控件，如何处理其它的文件格式，以及如何在你的应用程序中创建一个真实的 `HTML` 浏览器。

HTML解析器(parser)是如何工作的？

在 `wxPython` 中， `HTML` 窗口有它自己内在的解析器。实际上，这里有两个解析器类，但是其中的一个是另一个的改进。通常，使用解析器工作仅在你想扩展 `wx.html.HtmlWindow` 自身的功能时有用。如果你正在使用 `Python` 编程，并基于其它的目的想使用一个 `HTML` 解析器，那么我们建议你使用随同 `Python` 发布的 `htmllib` 和 `HTMLParser` 这两个解析器模块之一，或一个外部的 `Python` 工具如“ `Beautiful Soup` ”。

两个解析器类分别是 `wx.html.HtmlParser` ，它是一个更通用的解析器，另一个是 `wx.html.HtmlWinParser` ，它是 `wx.html.HtmlParser` 的子类，增加了对在 `wx.html.HtmlWindow` 中显示文本的支持。由于我们所关注的基本上是 `HTML` 窗口，所以我们将重点关注 `wx.html.HtmlWinParser` 。

要创建一个 `HTML` 解析器，可以使用两个构造函数之一。其中基本的一个是 `wx.html.HtmlWinParser()` ，没有参数。 `wx.html.HtmlWinParser` 的父类 `wx.html.HtmlParser` 也有一个没有参数的构造函数。你可以使用另一个构造函数 `wx.html.HtmlWinParser(wnd)` 将一个 `wx.html.HtmlWinParser()` 与一个已有的 `wx.html.HtmlWindow` 联系在一起，参数 `wnd` 是 `HTML` 窗口的实例。

要使用解析器，最简单的方法是调用 `Parse(source)` 方法。参数 `source` 是要被处理的 `HTML` 字符串。返回值是已解析了的数据。对于一个 `wx.html.HtmlWinParser` ，返回值是类 `wx.html.HtmlCell` 的一个实例。

`HTML` 解析器将 `HTML` 文本转换为一系列的单元，一个单元可以表示一些文本，一个图像，一个表，一个列表，或其它特定的元素。 `wx.html.HtmlCell` 的最重要的子类是 `wx.html.HtmlContainerCell` ，它是一个可以包含其它单元在其中的一个单元，如一个表或一个带有不同文本样式的段落。对于你解析的几乎任何文档，返回值都将是一个 `wx.html.HtmlContainerCell` 。每个单元都包含一个 `Draw(dc, x, y, view_y1, view_y2)` 方法，这使它可以在 `HTML` 窗口中自动绘制它的信息。

另一个重要的子类单元是 `wx.html.HtmlWidgetCell` ，它允许一个任意的 `wxPython` 控件像任何其它单元一样被插入到一个 `HTML` 文档中。除了可以包括用于格式化显示的静态文本，这也包括任何类型的用于管理 `HTML` 表单的控件。 `wx.html.HtmlWidgetCell` 的构造函数如下：

```
wx.html.HtmlWidgetCell(wnd, w=0)
```

其中参数 `wnd` 是要被绘制的 `wxPython` 控件。参数w是一个浮动宽度。如果w不为0，那么它应该是介于1和100之间的一个整数， `wnd` 控件的宽度则被动态地调整为相对于其父容器宽度的w%。

另外还有其它许多类型的用于显示 `HTML` 文档的部分的单元。更多的信息请参考 `wxWidget` 文档。

如何增加对新标记的支持？

被解析器返回的单元是被标记处理器内在的创建的，通过 `HTML` 标记，一个可插入的结构与 `HTML` 解析器单元的创建和处理相联系起来。你可以创建你自己的标记处理器，并将它与 `HTML` 标记相关联。使用这个机制，你可以扩展 `HTML` 窗口，以包括当前不支持的标准标记，或你自己发明的自定义的标记。图16.4显示了自定义 `HTML` 标记的用法。

图**16.4**



Figure 16.4
A wx.HtmlWindow using a custom tag handler

下例16.4是产生图16.4的代码。

例16.4 定义并使用自定义的标记处理器

```
import wx
import wx.html

page = """ html  body

This silly example shows how custom tags can be defined and used
 in a
wx.HtmlWindow.  We've defined a new tag,  blue  that will change
the  blue foreground color /blue  of the portions of the documen
t that
it encloses to some shade of blue.  The tag handler can also use
parameters specifed in the tag, for example:

 ul
 li   blue shade='sky' Sky Blue /blue
 li   blue shade='midnight' Midnight Blue /blue
 li   blue shade='dark' Dark Blue /blue
 li   blue shade='navy' Navy Blue /blue
 /ul

 /body  /html
"""
```

```
class BlueTagHandler(wx.html.HtmlWinTagHandler):#声明标记处理器
    def __init__(self):
        wx.html.HtmlWinTagHandler.__init__(self)

    def GetSupportedTags(self):#定义要处理的标记
        return "BLUE"

    def HandleTag(self, tag):#处理标记
        old = self.GetParser().GetActualColor()
        clr = "#0000FF"
        if tag.HasParam("SHADE"):
            shade = tag.GetParam("SHADE")
            if shade.upper() == "SKY":
                clr = "#3299CC"
            if shade.upper() == "MIDNIGHT":
                clr = "#2F2F4F"
            elif shade.upper() == "DARK":
                clr = "#00008B"
            elif shade.upper == "NAVY":
                clr = "#23238E"

        self.GetParser().SetActualColor(clr)
        self.GetParser().GetContainer().InsertCell(wx.html.HtmlC
olourCell(clr))

        self.ParseInner(tag)

        self.GetParser().SetActualColor(old)
        self.GetParser().GetContainer().InsertCell(wx.html.HtmlC
olourCell(old))

        return True

wx.html.HtmlWinParser_AddTagHandler(BlueTagHandler)

class MyHtmlFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title)
        html = wx.html.HtmlWindow(self)
        if "gtk2" in wx.PlatformInfo:
            html.SetStandardFonts()
        html.SetPage(page)

app = wx.PySimpleApp()
frm = MyHtmlFrame(None, "Custom HTML Tag Handler")
frm.Show()
app.MainLoop()
```

标记内在的由类 `wx.Html.Tag` 的方法来表现，标记的实例由 `HTML` 解析器来创建，通常，你不需要自己创建。表16.4显示了 `wx.Html.Tag` 类的方法，它们有用于检索标记的信息。

表**16.4** `wx.Html.Tag` 的一些方法

`GetAllParams()`	返回与标记相关的所有参数，返回值是一个字符串。出于某些目的，解析字符串比得到各个单独的参数更容易。
`GetName()`	以大写的方式，返回标记的名字。
`HasParam(param)`	如果标记给定了参数，则返回 `True` 。
`GetParam(param, with_commas=False)`	返回参数 `param` 的值。如果参数 `with_commas` 为 `True` ，那么你得到一个首尾都有引号的原始字符串。如果没有指定该参数，那么返回一个空字符串。方法 `GetParamAsColour(param)` 返回的参数值是一个 `wx.Color` ，方法 `GetParamAsInt(param)` 返回整数值。
`HasEnding()`	如果标记有结束标记的话，返回 `True` ，否则返回 `false` 。

用于扩展 `HTML` 窗口的标记处理器都是 `wx.html.HtmlWinTagHandler` 的子类。你的子类需要覆盖两个方法，并且你需要知道进一步的方法。需要覆盖的第一个方法是 `GetSupportedTags()` 。该方法返回由处理器管理的标记的列表。标记必需是大写的，并且标记之间以逗号分隔，中间不能有空格，如下所示：

```
GetSupportedTags(self):
    return "MYTAG,MYTAGPARAM"
```

第二个你需要覆盖的方法是 `HandleTag(tag)` 。在 `HandleTag(tag)` 方法中，你通过增加新的单元元素到解析器来处理标记（或者交替地改变解析器已经打开的容器单元）。你可以通过调用标记处理器的 `GetParser()` 方法来得到解析器。

要写一个 `HandleTag(tag)` 方法，你应该像下面这样做：

1、得到解析器。 2、对你的标记的参数做必要的处理，可能要改变或创建一个新的单元。 3、如果被解析的标记包括着内在的文本，那么解析标记之间的文本。 4、执行对于解析器所需要的任何清理工作。

如上所述，你使用 `GetParser()` 方法得解析器。要添加或编辑解析器中的单元，你有三个可选方案。第一个，如果你想添加另一个单元到容器中，你可以工作于当前的容器。第二个，你可以调用解析器的 `Container()` 方法，然后创建你的 `wx.html.HTMLCell` 子类实例，并通过调用容器的 `InsertCell(cell)` 方法将它添加到容器。

有时，你可能想在当前打开的容器中创建一个附属的或内嵌的容器。例如内嵌于表的一行中的一个单元格。要实现这个，你需要调用解析器的 `OpenContainer()` 方法。这个方法返回你的新的容器单元，你可以使用 `InsertCell(cell)` 方法来插入显示单元到你的新的容器单元中。对于每个在你的标记处理器中打开的容器，你应该使用 `CloseContainer()` 方法来关闭它。如果你没有成对的使用 `OpenContainer()` 和 `CloseContainer()` ，那么这将导致解析器解析余下的 `HTML` 文本时出现混乱。

第三个方案是创建一个与解析器的当前容器同级的容器，意思是不是嵌入的。例如一个新的段落——它不是前一段的一部分，也不附属于前一段；它是该页中?囊桓鲂碌氖堤濉N偃嗽诤馕鎏髦惺迪终飧鲋刍粲福奮阚枰後乇障执蜊娜萜鳎ɡ俅蛞桓鲂碌娜萜鳎菀缦滤◀᪤·荆?

```
parser = self.GetParser()
parser.CloseContainer()#关闭现存的容器
parser.OpenContainer()#打一个新的容器

# 添加或编辑解析器中的单元

parser.CloseContainer()
parser.OpenContainer()
```

如何支持其他的文件格式？

默认情况下， `HTML` 窗口可以处理带有 `MIME` 类型 `text` / `html`, `text` / `txt`, 和 `image` /*（假设 `wxPython` 图像处理器已经被装载）的文件。当碰上一个不是图像或 `HTML` 文件的文件时，该 `HTML` 窗口试图以纯文本的方式显示它。这可以不是你想要的行为。如果有一些文件你想以自定义的方式显示它的话，你可以创建一个 `wx.html.HtmlFilter` 来处理它。比如，你可能想以源代码树的方式显示 `XML` 文件，或使用语法着色来显示 `Python` 源代码文件。

要创建一个筛选器（ `filter` ），你必须建造 `wx.html.HtmlFilter` 的一个子类。 `wx.html.HtmlFilter` 类有两个方法，你必须都覆盖它们。这第一个方法是 `CanRead(file)` 。参数 `file` 是 `wx.FSFile` （一个打开的文件的 `wxPython` 表示）的一个实例。类 `wx.FSFile` 有两个属性，你可以用来决定你的筛选器是否能够读该文件。方法 `GetMimeType()` 以一个字符串的形式返回该文件的 `MIME` 类型。 `MIME` 类型通常由文件的后缀所定义。方法 `GetLocation()` 返回带有相关文件位置的绝对路径或 `URL` 的一个字符串。如果筛选器会处理该文件的话， `CanRead()` 方法应该返回 `True` ，否则返回 `False` 。处理 `Python` 源文件的 `CanRead()` 的一个示例如下：

```
CanRead(self, file):
return file.GetLocation().endswith('.py')
```

第二个你需要覆盖的方法是 `ReadFile(file)` 。这个方法要求一个同样的 `file` 参数，并返回该文件内容的一个字符串的 `HTML` 表达。如果你不想使用 `wxWidgets C` ++的文件机制来读该文件的话，你可以通过简单地打开位于 `file.GetLocation()` 的文件来使用 `Python` 的文件机制。

一旦筛选器被创建了，那么它必须被注册到 `wx.html.HtmlWindow` ，使用 `wx.html.HtmlWindow` 窗口的 `AddFilter(filter)` 静态方法来实现。参数 `filter` 是你的新的 `wx.html.HtmlFilter` 类的一个实例。一旦注册了筛选器，那么该窗口就可以使用筛选器来管理通过了 `CanRead()` 测试的文件对象。

如何得到一个性能更加完整的HTML控件？

尽管 `wx.html.HtmlWindow` 不是一个完整特性的浏览器面板，但是这儿有一对用于嵌入更加完整特性的 `HTML` 表现窗口的选择。如果你是在 `Windows` 平台上，你可以使用类 `wx.lib.iewin.IEHtmlWindow` ，它是 `Internet Explorer ActiveX` 控件的封装。这使得你能够直接将 `ie` 窗口嵌入到你的应用程序中。

使用 `IE` 控件比较简单，类似于使用内部的 `wxPython` 的 `HTML` 窗口。它的构造函数如下：

```
wx.lib.iewin.IEHtmlWindow(self, parent, ID=-1,
pos=wx.DefaultPosition, size=wx.DefaultSize, style=0,
name='IEHtmlWindow')
```

其中参数 `parent` 是父窗口， `ID` 是 `wxPython ID` 。对于 `IE` 窗口，这儿没有可用的样式标记。要装载 `HTML` 到 `IE` 组件中，可以使用方法 `LoadString(html)` ，其中参数 `html` 是要显示的一个 `HTML` 字符串。你可以使用方法 `LoadStream(stream)` 装载自一个打开的文件，或一个 `Python` 文件对象；或使用 `LoadString(URL)` 方法装载自一个 `URL` 。你能够使用 `GetText(asHTML)` 来获取当前显示的文本。参数 `asHTML` 是布尔值。如果为 `True` ，则返回 `HTML` 形式的文本，否则仅返回一个文本字符串。

在其它平台上，你可以尝试一下 `wxMozilla` 项目(`http://wxmozilla.sourceforge.net)` ，该项目尝试创建一个 `Mozilla Gecko` 表现器的 `wxPython` 封装。目前该项目仍在测试阶段。 `wxMozilla` 有用于 `Windows` 和 `Linux` 的安装包，对 `Mac OS X` 的支持正在开发中。

本章小结

1、 `HTML` 不再是只用于 `Internet` 了。在 `wxPython` 中，你可以使用一个 `HTML` 窗口来显示带有 `HTML` 标记的简单子集的文本。该 `HTML` 窗口属于 `wx.html.HtmlWindow` 类。除了 `HTML` 文本，该 `HTML` 窗口还可以管理任一的图像（图像处理器已装载的情况下）。

2、你可以让`HTML`窗口显示一个字符串，一个本地文件或一个`URL`的信息。你可以像通常的超文本浏览器的方式显示用户的敲击，或使用它自定义的响应。你也可以将`HTML`窗口与它的框架相连接起来，以便标题和状态信息自动地显示在适当的地方。`HTML`窗口维护着一个历史列表，你可以对它进行访问和处理。你可以使用类`wx.Html.HtmlEasyPrinting`来直接打印你的页面。

3、在`wxPython`中有一个`HTML`解析器，你可以用来创建用于你自己窗口的自定义标记。你也可以配置自定义的文件筛选器来在一个`HTML`窗口中表现其它的文件格式。

4、最后，如果你对`HTML`窗口的局限性不太满意的话，那么你可以使用一个对`IE`控件的`wxPython`封闭。如果你不在`Windows`上的话，这儿也有一个对`Mozilla Gecko HTML`表现器的`wxPython`的封装。

第十七章 wxPython的打印构架

本章内容

- 用 `wxPython` 打印
- 创建和显示打印对话框
- 创建和显示页面设置对话框
- 在你的程序中执行打印
- 执行一个打印预览

在第16章中，我们已经关注了 `wxPython` 的一打印方法：使用 `wx.HtmlEasyPrinting` 。如果你用该方法打印 `HTML` （或某些容易转换为 `HTML` 的文件）的话，这个方法将会工作的很好，但是要作为一个完善打印办法还是不够的。在 `wxPython` 中还有一个更为通用的打印构架，你可以用它来打印你想打印的任何东西。基本上，该构架使你能够使用设备上下文和绘制操作来执行打印。你也可以创建打印预览。

本章将讨论该构架中最重要的类： `wx.Printout` ，它管理实际的图片部分。打印输出实例可以由一个代表打印机的 `wx.Printer` 对象或用于打印预览的 `wx.PrintPreview` 对象来管理。多们也将讨论几个管理与打印相关的数据的类，以及用来显示信息给用户的对话框。

如何用wxPython打印？

我们将以类 `wx.Printout` 作为开始。首先你要创建你自定义的 `wx.Printout` 的子类。接着你要覆盖 `wx.Printout` 的方法以定义你自定义的打印行为。 `wx.Printout` 有7个你可以覆盖以自定义打印行为的方法。这些方法在一个打印会话过程期间被 `wxPython` 自动调用。图17.1其中的六个方法，它们被特定的事件触发。在大多数情况下，你不需要全部覆盖它们。

图**17.1**

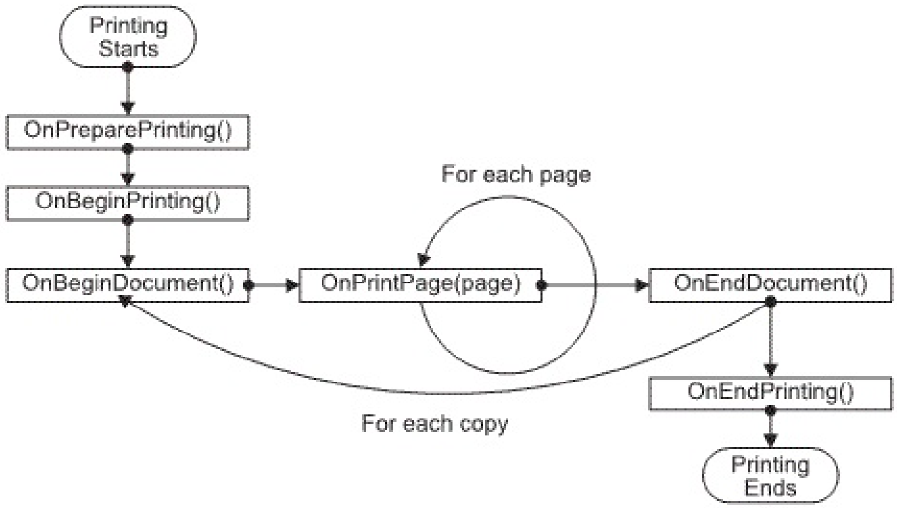


Figure 17.1 The lifecycle of a printout showing all the methods automatically called by wxPython

理解打印输出的生命周期

你通过创建一个你的打印输出对象的实例和一个类 `wx.Printer` 的实例启动一个打印会话：

```
wx.Printer(data=None)
```

可选的 `data` 参数是 `wx.PrintDialogData` 的一个实例。要开始实际的打印，需要调用 `wx.Printer` 的 `Print(parent, printout, prompt=True)` 方法。参数 `parent` 是父窗口（它被用作对话框的窗口中）。参数 `printout` 是你的 `wx.Printout` 实例。如果参数 `prompt` 为 `True` ，那么在打印之前， `wxPython` 将显示打印对话框，否则不显示。

在 `Print()` 方法开始后，它调用 `wx.Printout` 的第一个可被覆盖的方法 `OnPreparePrint()` 。 `OnPreparePrint()` 方法在 `wx.Printout` 实例做任何其它的事之前被确保调用，因此该方法是放置收集你的数据或做那些必须在打印开始之前所要做的计算的一个好的地方。实际的打印使用 `OnBeginPrinting()` 方法开始，你可以对该方法进行覆盖，以自定主你想要的行为——默认情况下，该方法什么也不做。 `OnBeginPrinting()` 在整个打印会话中只会被调用一次。

你希望打印的文档的每个单独的拷贝触发对 `OnBeginDocument(startPage, endPage)` 的一个调用，其中参数 `startPage, endPage` 告诉 `wxPython` 打印的起始页和最后一页。这两个参数都应该指定。如果你想覆盖这个方法，那么你必须调用它的基类的方法，因为基类的方法要做一些重要的工作（如调用 `wx.DC.StartDoc()` ）。在 `wxPython` 中，你可以使用 `base_OnBeginDocument(startPage, endPage)` 来调用其父类的方法。如果 `OnBeginDocument` 返回 `False` ，那么将取消打印工作。

你最有可能去覆盖的方法是 `OnPrintPage(pageNum)` ，该方法是你放置关于每一页的绘制命令的地方。参数 `pageNum` 是要打印的页的页码。在这个方法中，你调用 `GetDC()` ， `GetDC()` 根据你当前的系统平台返回一个适当的设备上下文。对于实际的打印，如果你是在一个微软的 `Windows` 系统上的话，那么 `GetDC()` 返回的是类 `wx.PrinterDC` 的实例。对于其它的系统，返回的是类 `wx.PostScriptDC` 的实例。如果你是处在一个打印预览操作中，那么对于任何的操作系统， `GetDC()` 返回的都是一个 `wx.MemoryDC` 。一旦你有了设备上下文，你就可以做你想做的设备上下文绘制操作，并且它们将被打印或预览。

在一个文档的副本打印结束后，一个 `OnEndDocument()` 调用被触发。另外，如果你要覆盖 `OnEndDocument()` 方法，那么你必须调用其基类的方法 `base_OnEndDocument()` 。 `base_OnEndDocument()` 将调用 `wx.DC.EndDoc()` 方法。当你的所有的副本被打印完后， `OnEndPrinting()` 方法被调用，这样就结束了打印会话。

`wx.Printout` 还有另一个可被覆盖的方法： `HasPage(pageNum)` 。该方法通常需要被覆盖，它被打印架构用于循环控制。如果参数 `pageNum` 存在于文档中，那么该方法返回 `True` ，否则返回 `False` 。

实战打印构架

下面我们将通过一个例子来展示打印构架实际上是如何工作的。这个例子由一个简单的用于打印文本文件的构架组成，并且应用程序让你能够键入简单的文本。图17.2显示了这个应用程序的结果。

图**17.1**

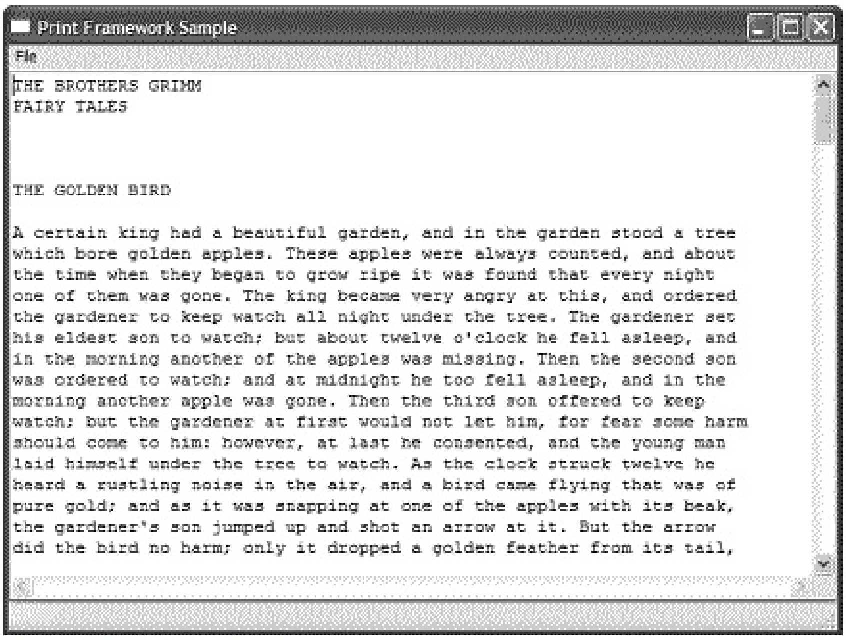


Figure 17.2 The simple printing framework in action

例17.1显示了我们已经讨论过的打印构架和我们将要接触的打印对话框机制。

例**17.1** 打印构架的一个较长的例子

```
import wx
import os

FONTSIZE = 10

class TextDocPrintout(wx.Printout):
    """
 A printout class that is able to print simple text documents.
 Does not handle page numbers o titles, and it assumes that no
 lines are longer than what will fit within the page width.  Tho
se
 features are left as an exercise for the reader. ;-)
 """
    def __init__(self, text, title, margins):
        wx.Printout.__init__(self, title)
        self.lines = text.split('\n')
        self.margins = margins

    def HasPage(self, page):
        return page  = self.numPages

    def GetPageInfo(self):
        return (1, self.numPages, 1, self.numPages)

    def CalculateScale(self, dc):
        # Scale the DC such that the printout is roughly the sam
e as
        # the screen scaling.
        ppiPrinterX, ppiPrinterY = self.GetPPIPrinter()
        ppiScreenX, ppiScreenY = self.GetPPIScreen()
        logScale = float(ppiPrinterX)/float(ppiScreenX)

        # Now adjust if the real page size is reduced (such as w
hen
        # drawing on a scaled wx.MemoryDC in the Print Preview.)
  If
        # page width == DC width then nothing changes, otherwise
 we
        # scale down for the DC.
        pw, ph = self.GetPageSizePixels()
        dw, dh = dc.GetSize()
        scale = logScale * float(dw)/float(pw)

        # Set the DC's scale.
        dc.SetUserScale(scale, scale)

        # Find the logical units per millimeter (for calculating
 the
        # margins)
        self.logUnitsMM = float(ppiPrinterX)/(logScale*25.4)

    def CalculateLayout(self, dc):
```

```
        # Determine the position of the margins and the
        # page/line height
        topLeft, bottomRight = self.margins
        dw, dh = dc.GetSize()
        self.x1 = topLeft.x * self.logUnitsMM
        self.y1 = topLeft.y * self.logUnitsMM
self.x2 = dc.DeviceToLogicalXRel(dw) - bottomRight.x * self.logU
nitsMM
self.y2 = dc.DeviceToLogicalYRel(dh) - bottomRight.y * self.logU
nitsMM

        # use a 1mm buffer around the inside of the box, and a f
ew
        # pixels between each line
        self.pageHeight = self.y2 - self.y1 - 2*self.logUnitsMM
        font = wx.Font(FONTSIZE, wx.TELETYPE, wx.NORMAL, wx.NORM
AL)
        dc.SetFont(font)
self.lineHeight = dc.GetCharHeight()
        self.linesPerPage = int(self.pageHeight/self.lineHeight)

    def OnPreparePrinting(self):
        # calculate the number of pages
        dc = self.GetDC()
        self.CalculateScale(dc)
        self.CalculateLayout(dc)
        self.numPages = len(self.lines) / self.linesPerPage
        if len(self.lines) % self.linesPerPage != 0:
            self.numPages += 1

    def OnPrintPage(self, page):
        dc = self.GetDC()
        self.CalculateScale(dc)
        self.CalculateLayout(dc)

        # draw a page outline at the margin points
        dc.SetPen(wx.Pen("black", 0))
        dc.SetBrush(wx.TRANSPARENT_BRUSH)
        r = wx.RectPP((self.x1, self.y1),
                      (self.x2, self.y2))
        dc.DrawRectangleRect(r)
        dc.SetClippingRect(r)

        # Draw the text lines for this page
        line = (page-1) * self.linesPerPage
        x = self.x1 + self.logUnitsMM
        y = self.y1 + self.logUnitsMM
        while line   (page * self.linesPerPage):
            dc.DrawText(self.lines[line], x, y)
            y += self.lineHeight
            line += 1
            if line  = len(self.lines):
                break
```

```
        return True

class PrintFrameworkSample(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, size=(640, 480),
                          title="Print Framework Sample")
        self.CreateStatusBar()

        # A text widget to display the doc and let it be edited
        self.tc = wx.TextCtrl(self, -1, "",
                              style=wx.TE_MULTILINE|wx.TE_DONTWR
AP)
        self.tc.SetFont(wx.Font(FONTSIZE, wx.TELETYPE, wx.NORMAL
, wx.NORMAL))
        filename = os.path.join(os.path.dirname(__file__), "samp
le-text.txt")
        self.tc.SetValue(open(filename).read())
        self.tc.Bind(wx.EVT_SET_FOCUS, self.OnClearSelection)
        wx.CallAfter(self.tc.SetInsertionPoint, 0)

        # Create the menu and menubar
        menu = wx.Menu()
        item = menu.Append(-1, "Page Setup...\tF5",
                           "Set up page margins and etc.")
        self.Bind(wx.EVT_MENU, self.OnPageSetup, item)
        item = menu.Append(-1, "Print Setup...\tF6",
                           "Set up the printer options, etc.")
        self.Bind(wx.EVT_MENU, self.OnPrintSetup, item)
        item = menu.Append(-1, "Print Preview...\tF7",
                           "View the printout on-screen")
        self.Bind(wx.EVT_MENU, self.OnPrintPreview, item)
        item = menu.Append(-1, "Print...\tF8", "Print the docume
nt")
        self.Bind(wx.EVT_MENU, self.OnPrint, item)
        menu.AppendSeparator()
        item = menu.Append(-1, "E ", "Close this application")
        self.Bind(wx.EVT_MENU, self.OnExit, item)

        menubar = wx.MenuBar()
        menubar.Append(menu, "  ")
        self.SetMenuBar(menubar)

        # initialize the print data and set some default values
        self.pdata = wx.PrintData()
        self.pdata.SetPaperId(wx.PAPER_LETTER)
        self.pdata.SetOrientation(wx.PORTRAIT)
        self.margins = (wx.Point(15,15), wx.Point(15,15))

    def OnExit(self, evt):
        self.Close()

    def OnClearSelection(self, evt):
        evt.Skip()
```

```
        wx.CallAfter(self.tc.SetInsertionPoint,
                     self.tc.GetInsertionPoint())

    def OnPageSetup(self, evt):
        data = wx.PageSetupDialogData()
        data.SetPrintData(self.pdata)

        data.SetDefaultMinMargins(True)
        data.SetMarginTopLeft(self.margins[0])
        data.SetMarginBottomRight(self.margins[1])

        dlg = wx.PageSetupDialog(self, data)
        if dlg.ShowModal() == wx.ID_OK:
            data = dlg.GetPageSetupData()
            self.pdata = wx.PrintData(data.GetPrintData()) # for
ce a copy
            self.pdata.SetPaperId(data.GetPaperId())
            self.margins = (data.GetMarginTopLeft(),
                            data.GetMarginBottomRight())
        dlg.Destroy()

    def OnPrintSetup(self, evt):
        data = wx.PrintDialogData(self.pdata)
        dlg = wx.PrintDialog(self, data)
        dlg.GetPrintDialogData().SetSetupDialog(True)
        dlg.ShowModal();
        data = dlg.GetPrintDialogData()
        self.pdata = wx.PrintData(data.GetPrintData()) # force a
 copy
        dlg.Destroy()

    def OnPrintPreview(self, evt):
        data = wx.PrintDialogData(self.pdata)
text = self.tc.GetValue()
        printout1 = TextDocPrintout(text, "title", self.margins)
        printout2 = None #TextDocPrintout(text, "title", self.ma
rgins)
        preview = wx.PrintPreview(printout1, printout2, data)
        if not preview.Ok():
            wx.MessageBox("Unable to create PrintPreview!", "Err
or")
        else:
            # create the preview frame such that it overlays the
 app frame
            frame = wx.PreviewFrame(preview, self, "Print Previe
w",
                                    pos=self.GetPosition(),
                                    size=self.GetSize())
            frame.Initialize()
            frame.Show()

    def OnPrint(self, evt):
        data = wx.PrintDialogData(self.pdata)
```

```
        printer = wx.Printer(data)
text = self.tc.GetValue()
        printout = TextDocPrintout(text, "title", self.margins)
        useSetupDialog = True
        if not printer.Print(self, printout, useSetupDialog) \
           and printer.GetLastError() == wx.PRINTER_ERROR:
            wx.MessageBox(
                "There was a problem printing.\n"
                "Perhaps your current printer is not set correct
ly?",
                "Printing Error", wx.OK)
        else:
            data = printer.GetPrintDialogData()
            self.pdata = wx.PrintData(data.GetPrintData()) # for
ce a copy
        printout.Destroy()

app = wx.PySimpleApp()
frm = PrintFrameworkSample()
frm.Show()
app.MainLoop()
```

例17.2中的打印输出类能够打印简单的文本文档，但是不能处理页码或标题，并且它创假设了行的宽度没有超过页面的宽度。对于例子的完善就留给读者作为一个练习。

上面最重要的代码片断是在构架的 `OnPreparePrinting()` 和 `OnPrintPage()` 以及示例窗口的 `OnPrint()` 方法中。

使用wx.Printout的方法工作

在 `wx.Printout` 中有几个 `get` *方法，它们使你能够获取当前打印环境的有关信息。表17.1列出了这些方法。

表17.1 **`wx.Printout`** 的信息获取方法

`GetDC()`	该方法返回关于打印机或打印预览的用于绘制文档的
`GetPageInfo()`	返回一个含有四个元素的元组(`minPage, maxPage, pageFrom, pageTo`) 。 `mir` 别是所允许的最小和最大页码，默认是1和32000。 `pageFrom, pageTo` 是必须被打印的范围，在你的子类中覆盖这个方法。
`GetPageSizeMM()`	返回包含一个页面的宽度和高度的一个(`w, h`) 元组
`GetPageSizePixels()`	返回一个页面的宽度和高度的一个(`w, h`) 元组，以打印输出被用于打印预览，那么像素数将反应当前的是说像素将会随缩放比列而变。
`GetPPIPrinter()`	返回当前打印机在垂直和水平方向上每英寸的像素的组。在预览中，这个值也是始终一致的，即使打印预了。
`GetPPIScreen()`	返回当前屏幕在垂直和水平方向上每英寸的像素的一在预览中，这个值也是始终一致的，即使打印预览的
`GetTitle()`	返回打印输出的标题。

在后面的几节中，我们将讨论如何呈现打印对话框给用户。

如何显示打印对话框？

诸如要打印那些面面，要打印多少副本这些关于打印工作的数据是由标准的打印对话框来管理的。打印对话框是与字体和颜色对话框类似的， `wxPython` 中的打印对话框实例仅仅是对本地控件和一个储存了对话框数据的分离的数据对象的简单封装。

创建一个打印对话框

图17.3显示了打印设置对话框的样例。

图**17.3**

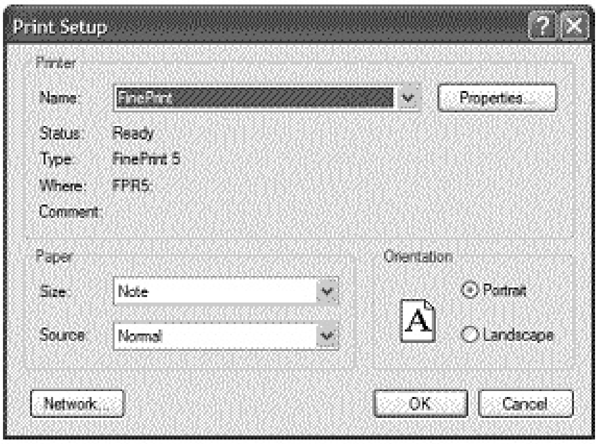


Figure 17.3
The print setup dialog

这里的对话框是类 `wx.PrintDialog` 的一个实例，你可以使用下面的构造函数来得到：

```
wx.PrintDialog(parent, data=None)
```

其中，参数 `parent` 是对话框的父框架，参数 `data` 是一个预先存在的 `wx.PrintDialogData` 实例，它用于对话框的初始数据。

使用方法

一旦你有了打印对话框，你就可以使用标准的 `ShowModal()` 方法来显示它， `ShowModal()` 方法将根据用户关闭对话框的方式而返回 `wx.ID_OK` 或 `wx.ID_CANCEL` 。 在你关闭了对话框之后，你可以使用 `GetPrintDialogData()` 方法来得到用户输入的数据。你也可以使用 `GetPrintDC()` 方法得到与数据相关联的打印机的设备上下文，如果还没有内容被创建，那么 `GetPrintDC()` 方法返回 `None` 。例17.1中的 `OnPrintSetup()` 方法显示了实际上对话框是如何被获取的。

使用属性

这个数据对象本身有几个属性，其中的一个是对 `wx.PrintData` 类型的一个对象的引用， `wx.PrintData` 有更多的属性。你可以使用构造函数 `wx.PrintDialogData()` 来创建你的 `wx.PrintDialogData` 对象。这使得你能够在打开对话框之前预设属性。

`wx.PrintDialogData` 对象有四个属性用于控制打印对话框的各个部分的有效性。方法 `EnableHelp(enable)` 用于开关帮助性能。至于对话框的其它部分， `EnablePageNumbers(enable)` 与页面数量输入框相关， `EnablePrintToFile(enable)` 管理实际的打印按钮， `EnableSelection(enable)` 在打印所有和仅打印被选项之间作切换。

表17.2显示了对话框数据对象的其它属性，它们使你能够管理有关打印请求的信息。

表17.2 `wx.PrintDialogData` 的属性

`GetAllPages()`	如果用户选择了打印整个文档这一选项，则返回 `True` 。
`SetCollate(flag)`	
`GetCollate()`	如果用户选择了核对打印的页，则返回 `True` 。
`SetFromPage(page)`	
`GetFromPage()`	如果用户选择从某一页打印，那么方法返回打印的第一页的整数页码。
`SetMaxPage(page)`	
`GetMaxPage()`	返回文档中最大的页码。
`SetMinPage(page)`	
`GetMinPage()`	返回文档中最小的页码。
`SetNoCopies()`	
`GetNoCopies()`	返回用户选择要打印的副本的数量。
`SetPrintData(printData)`	
`GetPrintData()`	返回与对话框相关联的 `wx.PrintData` 对象。
`SetPrintToFile(flag)`	
`GetPrintToFile()`	如果用户已经选择了打印到一个文件这一项，那么返回 `True` 。“打印到文件”这一机制由 `wxPython` 管理。
`SetSelection(flag)`	
`GetSelection()`	如果用户已经选择了只打印当前的选择这一项，那么返回 `True` 。
`SetToPage(page)`	
`GetToPage()`	如果用户指定了一个范围，那么返回打印的最后一页的页码。

被 `GetPrintData()` 方法返回的 `wx.PrintData` 实例提供了有关打印的更进一步的信息。通常这些信息是在你的打印对话框的打印设置子对话框中的。表17.3列出了 `wx.PrintData` 对象的属性。

表17.3 `wx.PrintData` 的属性

SetColour(flag)	
GetColour()	如果当前的打印是用于颜色打印的，那么返
SetDuplex(mode)	
GetDuplex()	返回当前关于双面打印的设置。值可以是 wx.DUPLEX_SIMPLE （非双面打印）， wx.DUPLEX_HORIZONTAL （横向双印）， wx.DUPLEX_VERTICAL （纵向双面
SetOrientation(orientation)	
GetOrientation()	返回纸张的打印定位（肖像或风景）。值可是 wx.LANDSCAPE 和 wx.PORTRAIT 。
SetPaperId(paperId)	
GetPaperId()	返回匹配纸张类型的标识符。通常的值有 wx.PAPER_LETTER, wx.PAPER_LEGAL, 完整的页面（纸张） ID 的列表见 wxWidg
SetPrinterName(printerName)	
GetPrinterName()	返回被系统引用的当前打印机的名字。如果么默认打印机被使用。
SetQuality(quality)	
GetQuality()	返回打印机的当前品质值。 set *方法仅接

如何显示页面设置对话框？

图17.4显示了页面设置对话框是如何让用户来设置与页面尺寸相关的数据的。

图**17.4**

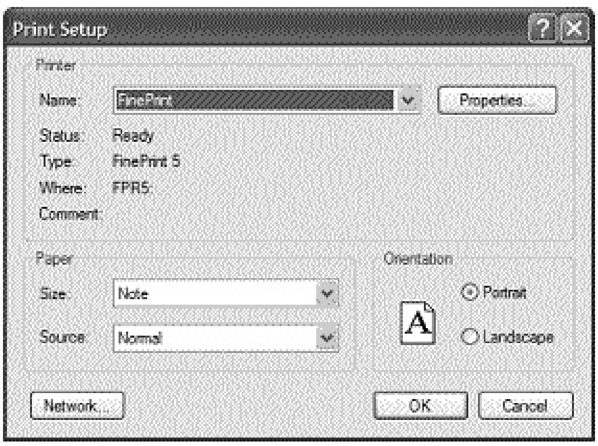


Figure 17.3
The print setup dialog

创建页面设置对话框

你可以通过实例化一个`wx.PageSetupDialog`类来创建一个页面设置对话框。

```
wx.PageSetupDialog(parent, data=None)
```

参数`parent`是新的对话框的父窗口。参数`data`是`wx.PageSetupDialogData`的一个实例默认为`None`。一旦页面设置对话框被创建了，那么这个对话框的行为就和其它任何模式对话框一样，并且你可以使用`ShowModal()`来显示它。通常，返回值表明了用户是否是使用`wx.ID_OK`或`wx.ID_CANCEL`按钮关闭的对话框窗口。在对话框关闭后，你可以通过调用`GetPageSetupDialogData()`来取得对数据对象的访问，`GetPageSetupDialogData()`返回类`wx.PageSetupDialogData`的一个实例。

使用页面设置属性工作

`wx.PageSetupDialogData`类有几个必须与页面设置一起使用的属性。表17.4展示了控制对话框自身显示的属性。除非有其它的指定，否则所有这些属性都默认为`True`。

表17.4 `wx.PageSetupDialogData`的对话框控制属性

`GetDefaultMinMargins()`	
`SetDefaultMinMargins(flag)`	如果这个属性为 `True` ，并且你是在微软的 Windows 系统上，那么页面设置将使用默认打印机的当前属性作为默认认的最小化页边距。否则，它将使用系统默认值。
`GetDefaultInfo()`	
`SetDefaultInfo(flag)`	如果这个属性为 `True` ，并且你是在微软的 Windows 系统上，那么这个页面设置对话框不会被显示。替而代之，当前打印机的所有默认值都将被放入数据对象。
`EnableHelp(flag)`	
`GetEnableHelp()`	如果为 `True` ，那么对话框的帮助部分是有效的。
`EnableMargins(flag)`	
`GetEnableMargins()`	如果为 `True` ，那么对话框的用于调整页边距的部分是有效的。
`EnableOrientation(flag)`	
`GetEnableOrientation()`	如果为 `True` ，那么对话框的用于改变页面定位的部分是有效的。
`EnablePaper(flag)`	
`GetEnablePaper()`	如果为 `True` ，那么对话框的用于允许用户改变页面（纸张）类型的部分是效的。
`EnablePrinter(flag)`	
`GetEnablePrinter()`	如果为 `True` ，那么允许用户设置打印机的按钮是有效的。

表17.5显示了 `wx.PageSetupDialogData` 类的附加的属性，这些属性用于控制页面的边距和尺寸。

表17.5 `wx.PageSetupDialogData` 的页边距和尺寸属性

GetMarginTopLeft()	
SetMarginTopLeft(pt)	get 方法返回一个 *wx.Point* ，其中的值*x*是当前的左边距，*y*是当前的上边距。 *set* 方法允许你使用一个 wx.Point 或一个 Python 元组来改变这些值。
GetMarginBottomRight()	
SetMarginBottomRight(pt)	get 方法返回一个 *wx.Point* ，其中的值*x*是当前的右边距，*y*是当前的下边距。 *set* 方法允许你使用一个 wx.Point 或一个 Python 元组来改变这些值。
GetMinMarginTopLeft()	
SetMinMarginTopLeft(pt)	同 GetMarginTopLeft() 中的一样，只是值是所允许的最小左边距和上边距。
GetMinMarginBottomRight()	
SetMinMarginBottomRight(pt)	同 GetMarginBottomRight() 中的一样，只是值是所允许的最小右边距和下边距。
GetPaperId()	
SetPaperId(id)	返回关于当前页面类型的 wxPython 标识符。同 wx.PrinterData 的属性。
GetPaperSize()	
SetPaperSize(size)	get *方法返回包含页面的水平和坚直方向尺寸的一个 wx.Size 实例。单位是毫米。
GetPrintData()	
SetPrintData(printData)	get *方法返回与当前打印会话相关的 wx.PrintData 实例。

到目前为止，我们已经讨论了所有关于数据对话框的整改，下面我们将重点放在打印上面。

如何打印？

到目前为止，我们已经见过了打印构架的所有部分，现是我们打印一些东西的时候了。实际的打印部分是由 wx.Printer 类的一个实例来控制的。与已经说明的其它部分相比，打印并不更简单。接下来，我们将对在例17.1中的 OnPrint() 中的

步骤作介绍。

第一步 按顺序得到你的所有数据

这至少应该包括带有打印机命令的 `wx.Printout` 对象，通常也要包括一个 `wx.PrintDialogData` 实例。

第二步 创建一个 `wx.Printer` 实例

创建该实例，要使用构造器 `wx.Printer(data=None)` 。可选参数 `data` 是一个 `wx.PrintDialogData` 实例。该数据控制打印，通常，你会想使用它。

第三步 使用 `wx.Printer` 的 `Print ()` 方法打印

`Print()` 方法如下：

```
Print(parent, printout, prompt=True)
```

其中参数 `parent` 是当打印时所触发的对话框的父窗口。 `printout` 是用于打印的 `wx.Printout` 对象。如果参数 `prompt` 为 `True` ，那么在打印之前显示打印对话框，否则将立即；启动打印。

如果打印成功，则 `Print()` 方法返回 `True` 。你能够调用 `GetLastError()` 方法来得到下列常量之一： `wx.PRINTER_CANCELLED` （如果失败是由于用户取消了打印所引起的）， `wx.PRINTER_ERROR` （如果失败在打印期间由打印自身所引起的），或 `wx.PRINTER_NO_ERROR` （如果 `Print()` 返回 `True` 且没有错误发生）。

这儿还有另外两个你可以使用一个 `wx.Printer` 实例做的事：

- 你可以使用 `CreateAbortWindow(parent,printout)` 来显示中止对话框，其中参数 `parent` 和 `printout` 同 `Print()` 方法中的。如果用户已经中止打印任务，你能够通过调用 `Abort()` 来发现，该方法在这种情况下返回 `True` 。
- 你可以使用 `PrintDialog(parent)` 来显式地显示打印对话框，并且你可以使用 `GetPrintDialogData()` 来得到活动的打印数据对象。

如何实现一个打印预览？

使用设备上下文的一个好处就是很容易管理打印预览，你可以使用一个屏幕设备上下文来代替打印机设备上下文。接下来的三个部分将讨论打印预览的过程。

第一步 创建预览实例

在一个打印预览中的第一步是创建类 `wx.PrintPreview` 的一个实例， `wx.PrintPreview` 类似 `wx.Printer` 。构造器如下：

```
wx.PrintPreview(printout, printoutForPrinting, data=None)
```

其中参数 `printout` 是一个 `wx.Printout` 对象，用于管理预览。参数 `printoutForPrinting` 是另一个 `wx.Printout` 对象。如果它不是 `None` ，那么当显示的时候，该打印预览窗口包含一 `Print` 按钮，该按钮启动打印。 `printoutForPrinting` 用于实际的打印。如果参数 `printoutForPrinting` 为 `None` ，那么 `Print` 按钮不显示。当然，你可以传递同一个实例或你的自定义打印输出类的相同版本的两个实例给参数 `printout` 和 `printoutForPrinting` 。参数 `data` 可以是一个 `wx.PrintData` 对象或一个 `wx.PrintDialogData` 对象。如果参数 `data` 指定了的话，那么它被用于控制该打印预览。在例17.1中，我们显示了一个在 `OnPrintPreview()` 方法中使用打印预览的例子。

第二步 创建预览框架

一旦你有了你的 `wx.PrintPreview` ，你就需要一框架以在其中观看你的 `wx.PrintPreview` 。该框架由类 `wx.PreviewFrame` 提供， `wx.PreviewFrame` 是 `wx.Frame` 的一个子类， `wx.Frame` 为预览提供基本的用户交互控件。 `wx.PreviewFrame` 的构造器如下：

```
wx.PreviewFrame(preview, parent, title, pos=wx.DefaultPosition,
size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE,
        name="frame")
```

其中真正有意义的参数是 `preview` ，它是要被预览的 `wx.PrintPreview` 实例。其它的参数都是标准的 `wx.Frame` 中的。 `wx.PreviewFrame` 不定义任何自定义的样式或事件。

第三步 初始化框架

在你显示你的 `wx.PreviewFrame` 之前，你需要调用 `Initialize()` 方法，该方法创建窗口的内部的部件并做其它的内部的计算。一旦你 `Show()` 了该框架，那么如果你想再改变预览窗口的感观，你可以使用考虑 `CreateControlBar()` 和 `CreateCanvas()` 方法，它们分别创建类 `wx.PreviewControlBar` 和 `wx.PreviewCanvas` 的对象。覆盖这些方法以创建你自己的画布(`canvas)` 和/或控制栏对象，使得你能够定制你的打印预览窗口的感观。

本章小结

1、这是 `wxPython` 中的一个通用的打印构架，它不仅可以打印 `HTML` ，还可以打印任何能够被绘制到设备上下文的东西。这个架构中的主要的类是 `wx.Printout` ，但是 `wx.Printer` 和 `wx.PrintPreview` 也是重要的。

2、`wx.Printout` 类管理图形打印的细节，并且它包含几个可以被覆盖来定制打印会话期间的行为和使用的数据的方法。打印发生在 `OnPrintPage()` 方法期间。

3、用于打印机设置和页面设置的标准的对话框是可以从 `wxPython` 来访问的。打印机设置对话框是 `wx.PrintDialog` 的一个实例，页面设置对话框是 `wx.PageSetupDialog` 的一个实例。这两个对话框都有相关的数据类，数据类使你的程序可以处理所有显示在对话框中的值。

4、一旦有了数据，那么实际将数据传送给打印机则是 `wx.Printer` 类的相对简单的应用。你可以使用 `wx.PrintPreview` 类来管理一个打印预览会话，该类包括一个打印预览框架，和根据该框架指定通常打印行为的选项。

第十八章 使用wxPython的其他功能

本章内容：

- 放置对象到剪贴板上
- 拖放
- 传送和获取自定义对象
- 使用 `wx.Timer` 设置定时的事件
- 编写多线程的 `wxPython` 应用程序

放置对象到剪贴板上

在 `wxPython` 中，剪贴板和拖放特性是紧密相关的。期间，内部窗口的通信是由使用 `wx.DataObject` 类或它的子类的一个实例作为中介的。 `wx.DataObject` 是一个特殊的数据对象，它包含描述输出数据格式的元数据。我们将从剪贴板入手，然后我们将讨论拖放的不同处理。

对于一个剪切和粘贴操作，有三个元素：

- `source(` 源)
- `clipboard(` 剪贴板)
- `target(` 目标)

如果 `source` 是在你的应用程序中，那么你的应用程序负责创建 `wx.DataObject` 的一个实例并把它放到剪贴板对象。通常 `source` 都是在你的应用程序的外部。

这里的 `clipboard` 是一个全局对象，它容纳数据并在必要时与操作系统的剪贴板交互。

`target` 对象负责从剪贴板获取 `wx.DataObject` 并把它转换为对你的应用程序有用的那一类数据。

得到剪贴板中的数据

如果你想你的应用程序能够引起一个剪贴事件，也就是说你想能够将数据剪切或复制到剪贴板，把数据放置到一个 `wx.DataObject` 里面。 `wx.DataObject` 知道自己能够被读写何种格式的数据。这点是比较重要的，例如如果你当时正在写一个词处理程序并希望给用户在粘贴时选择无格式文本的粘贴或丰富文本格式的粘贴的情况。然而大多数时候，在你的剪贴板行为中不需要太强大或太灵活的性能。对于最常用的情况， `wxPython` 提供了三个预定义的 `wx.DataObject` 的子类：纯文本，位图图像和文件名。

要传递纯文本，可以创建类`wx.TextDataObject`的一个实例，使用它如下的构造器：

```
wx.TextDataObject(text="")
```

参数`text`是你想传递到剪贴的文本。你可以使用`Text(text)`方法来设置该文本，你也可以使用`GetText()`方法来得到该文本，你还可以使用`GetTextLength()`方法来得到该文本的长度。

一旦你创建了这种数据对象后，接着你必须访问剪贴板。系统的剪贴板在`wxPython`中是一个全局性的对象，名为`wx.TheClipboard`。要使用它，可以使用它的`Open()`方法来打开它。如果该剪贴板被打开了则该方法返回`True`，否则返回`False`。如果该剪贴板正在被另一应用程序写入的话，该剪贴板的打开有可能会失败，因此在使用该剪贴板之前，你应该检查打开方法的返回值。当你使用完剪贴板之后，你应该调用它的`Close()`方法来关闭它。打开剪贴板会阻塞其它的剪贴板用户的使用，因此剪贴板打开的时间应该尽可能的短。

处理剪贴板中的数据

一旦你有了打开的剪贴板，你就可以处理它所包含的数据对象。你可以使用`SetData(data)`来将你的对象放置到剪贴板上，其中参数`data`是一个`wx.DataObject`实例。你可以使用方法`Clear()`方法来清空剪贴板。如果你希望在你的应用程序结束后，剪贴板上的数据还存在，那么你必须调用方法`Flush()`，该方法命令系统维持你的数据。否则，该`wxPython`剪贴板对象在你的应用程序退出时会被清除。

下面是一段添加文本到剪贴板的代码：

```
text_data = wx.TextDataObject("hi there")
if wx.TheClipboard.Open():
wx.TheClipboard.SetData(text_data)
    wx.TheClipboard.Close()
```

获得剪贴板中的文本数据

从剪贴板中获得文本数据也是很简单的。一旦你打开了剪贴板，你就可以调用`GetData(data)`方法，其中参数`data`是`wx.DataObject`的一些特定的子类的一个实例。如果剪贴板中的数据能够以与方法中的数据对象参数相一致的某种格式被输出的话，该方法的返回值则为`True`。这里，由于我们传递进的是一个`wx.TextDataObject`，那么返回值`True`就意味该剪贴板能够被转换到纯文本。下面是一段样例代码：

```
text_data = wx.TextDataObject()
if wx.TheClipboard.Open():
success = wx.TheClipboard.GetData(text_data)
wx.TheClipboard.Close()
if success:
return text_data.GetText()
```

注意，当你从剪贴板获取数据时，数据并不关心是哪个应用程序将它放置到剪贴板的。剪贴板中的数据本身被底层的操作系统所管理，`wxPython` 的责任是确保格式的匹配及你能够得到你能够处理的数据格式。

实战剪贴板

在这一节，我们将显示一个简单的例子，它演示了如何与剪贴板交换数据。它是一个有着两个按钮的框架，它使用户能够复制和粘贴文本。当你运行这个例子时，结果将会如图18.1所示。

图**18.1**

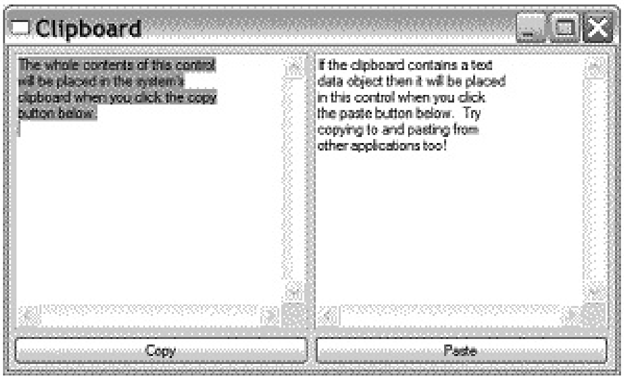


Figure 18.1
An example frame that uses the clipboard

例18.1是产生图18.1的代码。

例**18.1** 剪贴板交互示例

```
#-*- encoding:UTF-8 -*-
import wx

t1_text = """\
The whole contents of this control
will be placed in the system's
clipboard when you click the copy
button below.
"""

t2_text = """\
If the clipboard contains a text
data object then it will be placed
in this control when you click
the paste button below.  Try
```

```
copying to and pasting from
other applications too!
"""

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Clipboard",
                          size=(500,300))
        p = wx.Panel(self)

        # create the controls
        self.t1 = wx.TextCtrl(p, -1, t1_text,
                              style=wx.TE_MULTILINE|wx.HSCROLL)
        self.t2 = wx.TextCtrl(p, -1, t2_text,
                              style=wx.TE_MULTILINE|wx.HSCROLL)
        copy = wx.Button(p, -1, "Copy")
        paste = wx.Button(p, -1, "Paste")

        # setup the layout with sizers
        fgs = wx.FlexGridSizer(2, 2, 5, 5)
        fgs.AddGrowableRow(0)
        fgs.AddGrowableCol(0)
        fgs.AddGrowableCol(1)
        fgs.Add(self.t1, 0, wx.EXPAND)
        fgs.Add(self.t2, 0, wx.EXPAND)
        fgs.Add(copy, 0, wx.EXPAND)
        fgs.Add(paste, 0, wx.EXPAND)
        border = wx.BoxSizer()
        border.Add(fgs, 1, wx.EXPAND|wx.ALL, 5)
        p.SetSizer(border)

        # Bind events
        self.Bind(wx.EVT_BUTTON, self.OnDoCopy, copy)
        self.Bind(wx.EVT_BUTTON, self.OnDoPaste, paste)

    def OnDoCopy(self, evt):#Copy按钮的事件处理函数
        data = wx.TextDataObject()
        data.SetText(self.t1.GetValue())
        if wx.TheClipboard.Open():
            wx.TheClipboard.SetData(data)#将数据放置到剪贴板上
            wx.TheClipboard.Close()
        else:
            wx.MessageBox("Unable to open the clipboard", "Error
")

    def OnDoPaste(self, evt):#Paste按钮的事件处理函数
        success = False
        data = wx.TextDataObject()
        if wx.TheClipboard.Open():
            success = wx.TheClipboard.GetData(data)#从剪贴板得到数
据
            wx.TheClipboard.Close()
```

```
        if success:
            self.t2.SetValue(data.GetText())#更新文本控件
        else:
            wx.MessageBox(
                "There is no data in the clipboard in the requir
ed format",
                "Error")

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

获得剪贴板中的文本数据

从剪贴板中获得文本数据也是很简单的。一旦你打开了剪贴板，你就可以调用 `GetData(data)` 方法，其中参数 `data` 是 `wx.DataObject` 的一些特定的子类的一个实例。如果剪贴板中的数据能够以与方法中的数据对象参数相一致的某种格式被输出的话，该方法的返回值则为 `True` 。这里，由于我们传递进的是一个 `wx.TextDataObject` ，那么返回值 `True` 就意味该剪贴板能够被转换到纯文本。下面是一段样例代码：

```
text_data = wx.TextDataObject()
if wx.TheClipboard.Open():
success = wx.TheClipboard.GetData(text_data)
wx.TheClipboard.Close()
if success:
return text_data.GetText()
```

注意，当你从剪贴板获取数据时，数据并不关心是哪个应用程序将它放置到剪贴板的。剪贴板中的数据本身被底层的操作系统所管理， `wxPython` 的责任是确保格式的匹配及你能够得到你能够处理的数据格式。

实战剪贴板

在这一节，我们将显示一个简单的例子，它演示了如何与剪贴板交换数据。它是一个有着两个按钮的框架，它使用户能够复制和粘贴文本。当你运行这个例子时，结果将会如图18.1所示。

图**18.1**

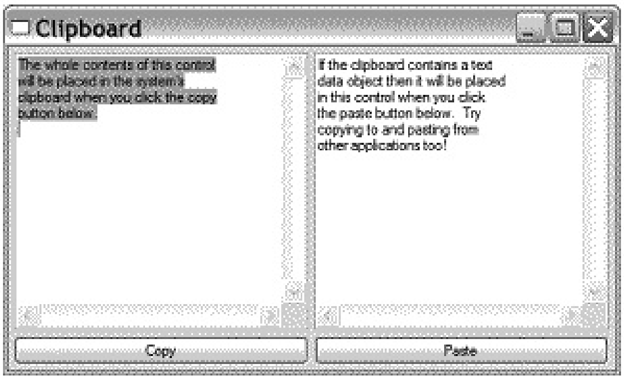


Figure 18.1
An example frame that uses the clipboard

例18.1是产生图18.1的代码。

例**18.1** 剪贴板交互示例

```
#-*- encoding:UTF-8 -*-
import wx

t1_text = """\
The whole contents of this control
will be placed in the system's
clipboard when you click the copy
button below.
"""

t2_text = """\
If the clipboard contains a text
data object then it will be placed
in this control when you click
the paste button below.  Try
copying to and pasting from
other applications too!
"""

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Clipboard",
                                  size=(500,300))
        p = wx.Panel(self)

        # create the controls
        self.t1 = wx.TextCtrl(p, -1, t1_text,
                                      style=wx.TE_MULTILINE|wx.HSCROLL)
        self.t2 = wx.TextCtrl(p, -1, t2_text,
                                      style=wx.TE_MULTILINE|wx.HSCROLL)
        copy = wx.Button(p, -1, "Copy")
        paste = wx.Button(p, -1, "Paste")

        # setup the layout with sizers
        fgs = wx.FlexGridSizer(2, 2, 5, 5)
        fgs.AddGrowableRow(0)
        fgs.AddGrowableCol(0)
```

传递其它格式的数据

```
        fgs.AddGrowableCol(1)
        fgs.Add(self.t1, 0, wx.EXPAND)
        fgs.Add(self.t2, 0, wx.EXPAND)
        fgs.Add(copy, 0, wx.EXPAND)
        fgs.Add(paste, 0, wx.EXPAND)
        border = wx.BoxSizer()
        border.Add(fgs, 1, wx.EXPAND|wx.ALL, 5)
        p.SetSizer(border)

        # Bind events
        self.Bind(wx.EVT_BUTTON, self.OnDoCopy, copy)
        self.Bind(wx.EVT_BUTTON, self.OnDoPaste, paste)

    def OnDoCopy(self, evt):#Copy按钮的事件处理函数
        data = wx.TextDataObject()
        data.SetText(self.t1.GetValue())
        if wx.TheClipboard.Open():
            wx.TheClipboard.SetData(data)#将数据放置到剪贴板上
            wx.TheClipboard.Close()
        else:
            wx.MessageBox("Unable to open the clipboard", "Error
")

    def OnDoPaste(self, evt):#Paste按钮的事件处理函数
        success = False
        data = wx.TextDataObject()
        if wx.TheClipboard.Open():
            success = wx.TheClipboard.GetData(data)#从剪贴板得到数
据
            wx.TheClipboard.Close()

        if success:
            self.t2.SetValue(data.GetText())#更新文本控件
        else:
            wx.MessageBox(
                "There is no data in the clipboard in the requir
ed format",
                "Error")

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

在下一节中，我们将讨论如何传递其它格式的数据，如位图。

传递其它格式的数据

经由剪贴板交互位图几乎与传递文本相同。你所使用的相关的数据对象子类是 `wx.BitmapDataObject` ，其 `get` 方法和 *`set`* 方法分别是 `GetBitmap()` 和 `SetBitmap(bitmap)` 。经由该数据对象与剪贴板交互的数据对象必须是 `wx.Bitmap` 类型的。

最后一个预定义的数据对象类型是 `wx.FileDataObject` 。通常该数据对象被用于拖放中（将在18.2节中讨论），例如当你将一个文件从你的资源管理器或查找窗口放置到你的应用程序上时。你可以使用该数据对象从剪贴板接受文件名数据，并且你可以使用方法 `GetFilenames()` 来从该数据对象获取文件名，该方法返回一个文件名的列表，列表中的每个文件名是已经被添加到剪贴板的文件名。你可以使用该数据对象的 `AddFile(file)` 方法来将数据放置到剪贴板上，该方法将一个文件名字符串添加到该数据对象。这里没有其它的方法用于直接处理列表，所以这就要靠你自己了。本章的稍后部份，我们将讨论如何经由剪贴板传送自定义对象，以及如何拖放对象。

拖放源

拖放是一个类似剪切和粘贴的功能。它是在你的应用程序的不同部分之间或两个不同的应用程序之间传送数据。由于管理数据和格式几乎是相同的，所以 `wxPython` 同样使用 `wx.DataObject` 族来确保对格式作恰当的处理。

拖放和剪切粘贴的最大不同是，剪切粘贴信赖于中介剪贴板的存在。因为是剪贴板管理数据，所以源程序将数据传送后就不管之后的事情了。这对于拖放却不然，源应用程序不仅虽要创建一个拖动管理器来服务于剪贴板，而且它也必须等待目标应用程序的响应。不同于一个剪贴板的操作，在拖放中，是目标应用来决定操作是一个剪贴或拷贝，所以源应用必须等待以确定传送的数据所用的目的。

通常，对源的拖动操作是在一个事件处理函数中进行，通常是一个鼠标事件，因为拖动通常都随鼠标的按下事件发生。创建一个拖动源要求四步：

1、创建数据对象 2、创建 `wx.DropSource` 实例 3、执行拖动操作 4、取消或允许释放

步骤**1** 创建一个数据对象

这第一步是创建你的数据对象。这在早先的剪贴板操作中有很好的说明。对于简单的数据，使用预定义的 `wx.DataObject` 的子类是最简单的。有了数据对象后，你可以创建一个释放源实例

步骤**2** 创建释放源实例

接下来的步骤是创建一个 `wx.DropSource` 实例，它扮演类似于剪贴板这样的传送角色。 `wx.DropSource` 的构造函数如下：

```
wx.DropSource(win, iconCopy=wx.NullIconOrCursor,
iconMove=wx.NullIconOrCursor,
        iconNone=wx.NullIconOrCursor)
```

参数`win`是初始化拖放操作的窗口对象。其余的三个参数用于使用自定义的图片来代表鼠标的拖动意义（拷贝、移动、取消释放）。如果这三个参数没有指定，那么使用系统的默认值。在微软的`Windows`系统上，图片必须是`wx.Cursor`对象，对于`Unix`则应是`wx.Icon`对象—— `Mac OS`目前忽略你的自定义图片。

一旦你有了你的`wx.DropSource`实例，那么就可以使用方法`SetData(data)`来将你的数据对象关联到`wx.DropSource`实例。接下来我们将讨论实际的拖动。

步骤**3** 执行拖动

拖动操作通过调用释放源的方法`DoDragDrop(flags=wx.Drag_CopyOnly)`来开始。参数`flags`表示目标可对数据执行的何种操作。取值有`wx.Drag_AllowMove`，它表示批准执行一个移动或拷贝，`wx.Drag_DefaultMove`表示不仅允许执行一个移动或拷贝，而且做默认的移动操作，`wx.Drag_CopyOnly`表示只执行一个拷贝操作。

步骤**4** 处理释放

`DoDragDrop()`方法直到释放被目标取消或接受才会返回。在此期间，虽然绘制事件会继续被发送，但你的应用程序的线程被阻塞。`DoDragDrop()`的返回值基于目标所要求的操作，取值如下：

`wx.DragCancel`（对于取消操作而言）

`wx.DragCopy`（对于拷贝操作而言）

`wx.DragMove`（对于移动操作而言）

`wx.DragNone`（对于错误而言）

对这些返回值的响应由你的应用程序来负责。通常对于响应移动要删除被拖动的数据外，对于拷贝则是什么也不用做。

实战拖动

例18.2显示了一个完整的拖动源控件，适合于通过拖动上面的箭头图片到你的系统的任何接受文本的应用上（如`Microsoft word`）。图18.2图示了这个例子。

图**18.2**

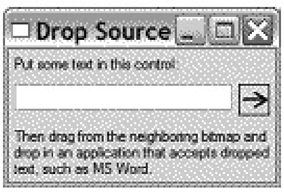


Figure 18.2 The drop source control as it looks on screen

例**18.2** 一个小的拖动源控件

```
#-*- encoding:UTF-8 -*-

import wx

class DragController(wx.Control):
    """
 Just a little control to handle dragging the text from a text
 control.  We use a separate control so as to not interfere with
 the native drag-select functionality of the native text control
.
 """
    def __init__(self, parent, source, size=(25,25)):
        wx.Control.__init__(self, parent, -1, size=size,
                            style=wx.SIMPLE_BORDER)
        self.source = source
        self.SetMinSize(size)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)

    def OnPaint(self, evt):
        # draw a simple arrow
        dc = wx.BufferedPaintDC(self)
        dc.SetBackground(wx.Brush(self.GetBackgroundColour()))
        dc.Clear()
        w, h = dc.GetSize()
        y = h/2
        dc.SetPen(wx.Pen("dark blue", 2))
        dc.DrawLine(w/8,   y,  w-w/8, y)
        dc.DrawLine(w-w/8, y,  w/2,   h/4)
        dc.DrawLine(w-w/8, y,  w/2,   3*h/4)

    def OnLeftDown(self, evt):
        text = self.source.GetValue()
        data = wx.TextDataObject(text)
        dropSource = wx.DropSource(self)#创建释放源
        dropSource.SetData(data)#设置数据
        result = dropSource.DoDragDrop(wx.Drag_AllowMove)#执行释
放

        # if the user wants to move the data then we should dele
te it
        # from the source
        if result == wx.DragMove:
            self.source.SetValue("")#如果需要的话，删除源中的数据

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Drop Source")
        p = wx.Panel(self)

        # create the controls
        label1 = wx.StaticText(p, -1, "Put some text in this con
```

```
trol:")
        label2 = wx.StaticText(p, -1,
           "Then drag from the neighboring bitmap and\n"
           "drop in an application that accepts dropped\n"
           "text, such as MS Word.")
        text = wx.TextCtrl(p, -1, "Some text")
        dragctl = DragController(p, text)

        # setup the layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(label1, 0, wx.ALL, 5)
        hrow = wx.BoxSizer(wx.HORIZONTAL)
        hrow.Add(text, 1, wx.RIGHT, 5)
        hrow.Add(dragctl, 0)
        sizer.Add(hrow, 0, wx.EXPAND|wx.ALL, 5)
        sizer.Add(label2, 0, wx.ALL, 5)
        p.SetSizer(sizer)
        sizer.Fit(self)

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

接下来，我们将给你展示目标处的拖放。

拖放到的目标

实现拖放到的目标的步骤基本上借鉴了实现拖放源的步骤。其中最大的区别是，实现拖放源，你可以直接使用类 `wx.DropSource` ，而对于目标，你首先必须写你的自定义的 `wx.DropTarget` 的子类。一旦你有了你的目标类，你将需要创建它的一个实例，并通过使用 `wx.Window` 的 `SetDropTarget(target)` 方法将该实例与任一 `wx.Window` 的实例关联起来。设置了目标后， `wx.Window` 的实例（不论它是一个窗口，一个按钮，一个文本域或其它的控件）就变成了一个有效的释放目标。为了在你的释放目标上接受数据，你也必须创建一个所需要类型的 `wx.DataObject` 对象，并使用释放目标方法 `SetDataObject(data)` 将 `wx.DataObject` 对象与释放目标关联起来。在实际释放操作前，你需要预先定义数据对象，以便该释放目标能够正确地处理格式。要从目标获取该数据对象，有一个方法 `GetDataObject()` 。下面的样板代码使得释放目标能够接受文本（仅能接受文本）。这是因为数据对象已经被设置为 `wx.TextDataObject` 的一个实例。

```
class MyDropTarget(wx.DropTarget):
def __init__(self):
self.data = wx.TextDataObject()
self.SetDataObject(data)
target = MyDataTarget()
win.SetDropTarget(target)
```

使用你的释放到的目标

当一个释放发生时，你的 `wx.DropTarget` 子类的各种事件函数将被调用。其中最重要的是 `OnData(x, y, default)` ，它是你必须在你自定义的释放目标类中覆盖的一个事件方法。参数 `x,y` 是释放时鼠标的位置。 `default` 参数是 `DoDragDrop()` 的四个取值之一，具体的值基于操作系统，传递给 `DoDragDrop()` 标志和当释放发生时修饰键的状态。在且仅在 `OnData()` 方法中，你可以调用 `GetData()` 。 `GetData()` 方法要求来自释放源的实际的数据并把它放入与你的释放目标对象相关联的数据对象中。 `GetData()` 不返回数据对象，所以你通常应该用一个实例变量来包含你的数据对象。下面是关于 `MyDropTarget.OnData()` 的样板代码：

```
def OnData(self, x, y, default):
self.GetData()
actual_data = self.data.GetText()
    # Do something with the data here...
return default
```

`OnData()` 的返回值应该是要导致操作——你应该返回参数 `default` 的值，除非这儿有一个错误并且你需要返回 `wx.DragNone` 。一旦你有了数据，你就可以对它作你想做的。记住，由于 `OnData()` 返回的是关于所导致操作的相关信息，而非数据本身，所以如果你想在别处使用该数据的话，你需要将数据放置在一个实例变量里面（该变量在该方法外仍然可以被访问）。

在释放操作完成或取消后，返回自 `OnData()` 的导致操作类型的数据被从 `DoDragDrop()` 的返回，并且释放源的线程将继续进行。

在 `wx.DropTarget` 类中有五个 `On...` 方法，你可以在你的子类中覆盖它们以在目标被调用时提供自定义的行为。我们已经见过了其中的 `OnData()` ，另外的如下：

`OnDrop(x, y)` `OnEnter(x, y, default)` `OnDragOver(x, y, default)` `OnLeave()`

其中的参数 `x, y, default` 同 `OnData()` 。你不必覆盖这些方法，但是如果你想在你的应用程序中提供自定义的功能的话，你可以覆盖这些方法。

当鼠标进入释放到的目标区域时， `OnEnter()` 方法首先被调用。你可以使用该方法来更新一个状态窗口。该方法返回如果释放发生时要执行的操作（通常是 `default` 的值）或 `wx.DragNone` （如果你不接受释放的话）。该方法的返回值被 `wxPython` 用来指定当鼠标移动到窗口上时，哪个图标或光标被用作显示。当鼠标位于窗口中时，方法 `OnDragOver()` 接着被调用，它返回所期望的操作或 `wx.DragNone` 。当鼠标被释放并且释放(`drop)` 发生时， `OnDrop()` 方法被调用，并且它默认调用 `OnData()` 。最后，当光标退出窗口时 `OnLeave()` 被调用。

与数据对象一同， `wxPython` 提供了两个预定义的释放到的目标类来涵盖最常见的情况。除了在这些情况中预定义的类会为你处理 `wx.DataObject` ，你仍然需要创建一个子类并覆盖一个方法来处理相关的数据。关于文本，类 `wx.TextDropTarget` 提供了可覆盖的方法 `OnDropText(x, y, data)` ，你将使用通过覆盖该方法来替代覆盖 `OnData()` 。参数 `x,y` 是释放到的坐标，参数 `data` 是被释放的字符串，该字符串你可以立即使用而不用必须对数据对象作更多的查询。如果你接受新的文本的话，你的覆盖应该返回 `True` ，否则应返回 `False` 。对于文件的释放，相关的预定义的类是 `wx.FileDropTarget` ，并且可覆盖的方法是 `OnDropFiles(x, y, filenames)` ，参数 `filenames` 是被释放的文件的名字的一个列表。另外，必要的时候你可以处理它们，当完成时可以返回 `True` 或 `False` 。

实战释放

例18.3中的代码显示了如何创建一个框架（窗口）用以接受文件的释放。你可以通过从资源管理器或查找窗口拖动一个文件到该框架（窗口）上来测试例子代码，并观查显示在窗口中的关于文件的信息。图18.3是运行后的结果。

图**18.3**

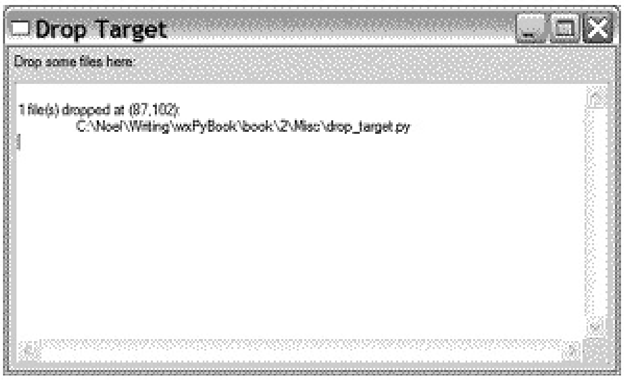


Figure 18.3
The drop target frame

例**18.3** 文件释放到的目标的相关代码

```
#-*- encoding:UTF-8 -*-
import wx

class MyFileDropTarget(wx.FileDropTarget):#声明释放到的目标
    def __init__(self, window):
        wx.FileDropTarget.__init__(self)
        self.window = window

    def OnDropFiles(self, x, y, filenames):#释放文件处理函数数据
        self.window.AppendText("\n%d file(s) dropped at (%d,%d):
\n" %
                               (len(filenames), x, y))
        for file in filenames:
            self.window.AppendText("\t%s\n" % file)

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Drop Target",
                          size=(500,300))
        p = wx.Panel(self)

        # create the controls
        label = wx.StaticText(p, -1, "Drop some files here:")
        text = wx.TextCtrl(p, -1, "",
                           style=wx.TE_MULTILINE|wx.HSCROLL)

        # setup the layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(label, 0, wx.ALL, 5)
        sizer.Add(text, 1, wx.EXPAND|wx.ALL, 5)
        p.SetSizer(sizer)

        # make the text control be a drop target
        dt = MyFileDropTarget(text)#将文本控件作为释放到的目标
        text.SetDropTarget(dt)

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

到目前为止，我们还是局限于对 `wxPython` 的预定义的对象的数据传送的讨论。接下来，我们将讨论如何将你自己的数据放到剪贴板上。

传送自定义对象

使用 `wxPython` 的预定义的数据对象，你只能工作于纯文本、位图或文件。而更有创建性的是，你应该让你自定义的对象能够在应用之间被传送。在这一节，我将给你展示如何给你的 `wxPython` 应用程序增加更高级的性能，如传送自定义的数据对

象和以多种格式传送一个对象。

传送自定义的数据对象

尽管文本、位图的数据对象和文件名的列表对于不同的使用已经足够了，但有时你仍然需要传送自定义的对象，如你自己的图形格式或一个自定义的数据结构。接下来，在保留对你的对象将接受的数据的类型的控制时，我们将涉及传送自定义数据对象的机制。该方法的局限是它只能工作在 `wxPython` 内，你不能使用这个方法来让其它的应用程序去读你的自定义的格式。要将 `RTF` （丰富文本格式）传送给 `Microsoft Word` ，该机制将不工作。

要实现自定义的数据传送，我们将使用 `wxPython` 的类 `wx.CustomDataObject` ，它被设计来用于处理任意的数据。 `wx.CustomDataObject` 的构造器如下：

```
wx.CustomDataObject(format=wx.FormatInvalid)
```

参数 `format` 技术上应该是类 `wx.DataFormat` 的一个实例，但为了我们的目的，我们可以只给它传递一个字符串，数据类型的责任由 `wxPython` 来考虑。我们只需要这个字符串作为自定义格式的一个标签，以与其它的区分开来。一旦我们有了我们自定义的数据实例，我们就可以使用方法 `SetData(data)` 将数据放入到自定义的数据实例中。参数 `data` 必须是一个字符串。下面是一段样板代码：

```
data_object = wx.CustomDataObject("MyNiftyFormat")
data = cPickle.dumps(my_object)
data_object.SetData(data)
```

在这段代码片断之后，你可以将 `data_object` 传递到剪贴板或另一个数据源，以继续数据的传送。

得到自定义对象

要得到该对象，需要执行相同的基本步骤。对于从剪贴板获取，先创建相同格式的一个自定义数据对象，然后得到数据并对得到的数据进行逆 `pickle` 操作（ `pickle` 有加工的意思）。

```
data_object = wx.CustomDataObject("MyNiftyFormat")
if wx.TheClipboard.Open():
success = wx.TheClipboard.GetData(data_object)
wx.TheClipboard.Close()
if success:
pickled_data = data_object.GetData()
object = cPickle.loads(pickled_data)
```

拖放工作是类似的。使用已 `pickle` 的数据设置释放源的数据对象，并将设置的数据对象给你的自定义的数据对象，数据的目标在它的 `OnData()` 方法中对数据进行逆 `pickle` 操作并把数据放到有用的地方。

创建自定义对象的另一个方法是建造你自己的 `wx.DataObject` 子类。如果你选择这条途径，那么你会希望实现你自己的诸如 `wx.PyDataObjectSimple` （用于通常的对象），
或 `wx.PyTextDataObject` ， `wx.PyBitmapDataObject,` 或 `wx.PyFileDataObje` 的一个子类。这将使你能够覆盖所有必要的方法。

以多种格式传送对象

使用 `wxPython` 的数据对象来用于数据传送的最大好处是，数据对象了解数据格式。一个数据对象甚至能够用多种的格式来管理相同的数据。例如，你可能希望你自己的应用程序能够接受你的自定义的文本格式对象的数据，但是你仍然希望其它的应用能够以纯文本的格式接受该数据。

管理该功能的机制是类 `wx.DataObjectComposite` 。目前，我们所见过的所有被继承的数据对象都是 `wx.DataObjectSimple` 的子类。 `wx.DataObjectComposite` 的目的是将任意数量的简单数据对象合并为一个数据对象。该合并后的对象能够将它的数据提供给与构成它的任一简单类型匹配的一个数据对象。

要建造一个合成的数据对象，首先要使用一个无参的构造器 `wx.DataObjectComposite()` 作为开始，然后使用 `Add(data, preferred=False)` 分别增加简单数据对象。要建造一个合并了你的自定义格式和纯文本的数据对象，可以如下这样：

```
data_object = wx.CustomDataObject("MyNiftyFormat")
data_object.SetData(cPickle.dumps(my_object))
text_object = wx.TextDataObject(str(my_object))
composite = wx.DataObjectComposite()
composite.Add(data_object)
composite.Add(text_object)
```

此后，将这个合成的对象传递给剪贴板或你的释放源。如果目标类要求一个使用了自定义格式的对象，那么它接受已 `pickle` 的对象。如果它要求纯文本的数据，那么它得到字符串表达式。

下节内容：我们将给你展示如何使用一个定时器来管理定时事件。

使用wx.Timer来设置定时事件

有时你需要让你的应用程序产生基于时间段的事件。要得到这个功能，你可以使用类 `wx.Timer` 。

产生EVT_TIMER事件

对 `wx.Timer` 最灵活和最有效的用法是使它产生 `EVT_TIMER` ，并将该事件如同其它事件一样进行绑定。

创建定时器

要创建一个定时器，首先要使用下面的构造器来创建一个 `wx.Timer` 的实例。

```
wx.Timer(owner=None, id=-1)
```

其中参数 `owner` 是实现 `wx.EvtHandler` 的实例，即任一能够接受事件通知的 `wxPython` 控件或其它的东西。参数 `id` 用于区分不同的定时器。如果没有指定 `id` ，则 `wxPython` 会为你生成一个 `id` 号。如果当你创建定时器时，你不想设置参数 `owner` 和 `id` ，那么你可以以后随时使用 `SetOwner(owner=None, id=` -1)方法来设置，它设置同样的两个参数。

绑定定时器

在你创建了定时器之后，你可以如下面一行的代码来在你的事件处理控件中绑定 `wx.EVT_TIMER` 事件。

```
self.Bind(wx.EVT_TIMER, self.OnTimerEvent)
```

如果你需要绑定多个定时器到多个处理函数，你可以给 `Bind` 函数传递每个定时器的 `ID` ，或将定时器对象作为源参数来传递。

```
timer1 = wx.Timer(self)
timer2 = wx.Timer(self)
self.Bind(wx.EVT_TIMER, self.OnTimer1Event, timer1)
self.Bind(wx.EVT_TIMER, self.OnTimer2Event, timer2)
```

启动和停止定时器

在定时器事件被绑定后，你所需要做的所有事情就是启动该定时器，使用方法 `Start(milliseconds=` -1, `oneShot=False)` 。其中参数 `milliseconds` 是毫秒数。这将在经过 `milliseconds` 时间后，产生一个 `wx.EVT_TIMER` 事件。如果 `milliseconds=` -1，那么将使用早先的毫秒数。如果 `oneShot` 为 `True` ，那么定时器只产生 `wx.EVT_TIMER` 事件一次，然后定时器停止。否则，你必须显式地使用 `Stop()` 方法来停止定时器。 例18.4使用了定时器机制来驱动一个数字时钟，并每秒刷新一次显示。

例**18.4** 一个简单的数字时钟

```
#-*- encoding:UTF-8 -*-
import wx
import time

class ClockWindow(wx.Window):
    def __init__(self, parent):
        wx.Window.__init__(self, parent)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.timer = wx.Timer(self)#创建定时器
        self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)#绑定一
个定时器事件
        self.timer.Start(1000)#设定时间间隔

    def Draw(self, dc):#绘制当前时间
        t = time.localtime(time.time())
        st = time.strftime("%I:%M:%S", t)
        w, h = self.GetClientSize()
        dc.SetBackground(wx.Brush(self.GetBackgroundColour()))
        dc.Clear()
        dc.SetFont(wx.Font(30, wx.SWISS, wx.NORMAL, wx.NORMAL))
        tw, th = dc.GetTextExtent(st)
        dc.DrawText(st, (w-tw)/2, (h)/2 - th/2)

    def OnTimer(self, evt):#显示时间事件处理函数
        dc = wx.BufferedDC(wx.ClientDC(self))
        self.Draw(dc)

    def OnPaint(self, evt):
        dc = wx.BufferedPaintDC(self)
        self.Draw(dc)

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="wx.Timer")
        ClockWindow(self)

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

确定当前定时器的状态

你可以使用方法 `IsRunning()` 来确定定时器的当前状态，使用方法 `GetInterval()` 来得到当前的时间间隔。如果定时器正在运行并且只运行一次的话，方法 `IsOneShot()` 返回 `True` 。

`wx.TimerEvent` 几乎与它的父类 `wx.Event` 是一样的，除了它不包括 `wx.GetInterval()` 方法来返回定时器的时间间隔外。万一你将来自多个定时器的事件绑定给了相同的处理函数，并希望根据特定的定时器的事件来做不同的动

作的话，可使用事件方法`GetId()`来返回定时器的`ID`，以区别对待。

学习定时器的其它用法

另一种使用定时器的方法是子类化`wx.Timer`。在你的子类中你可以覆盖方法`Notify()`。在父类中，该方法每次在定时器经过指定的时间间隔后被自动调用，它触发定时器事件。然而你的子类没有义务去触发一个定时器事件，你可以在该`Notify()`方法中做你想做的事，以响应定时器的时间间隔。

要在未来某时触发一个特定的行为，有一个被称为`wx.FutureCall`的类可以使用。它的构造器如下：

```
wx.FutureCall(interval, callable, *args, **kwargs)
```

一旦它被创建后，`wx.FutureCall`的实例将等待`interval`毫秒，然后调用传递给参数`callable`的对象，参数`args,`* `kwargs`是`callable`中的对象所要使用的。`wx.FutureCall`只触发一次定时事件。

下节内容提示：创建一个多线程的`wxPython`应用程序

创建一个多线程的wxPython应用程序

在大多数的`GUI`应用程序中，在应用程序的后台中长期执行一个处理过程而不干涉用户与应用程序的其它部分的交互是有好处的。允许后台处理的机制通常是产生一个线程并在该线程中长期执行一个处理过程。对于`wxPython`的多线程，在这一节我们有两点需要特别说明。

最重要的一点是，`GUI`的操作必须发生在主线程或应用程序的主循环所处的地方中。在一个单独的线程中执行`GUI`操作对于无法预知的程序崩溃和调试来说是一个好的办法。基于技术方面的原因，如许多`Unix`的`GUI`库不是线程安全性的，以及在微软`Windows`下`UI`对象的创建问题，`wxPython`没有设计它自己的发生在多线程中的事件，所以我们建议你也不要尝试。

上面的禁令包括与屏幕交互的任何项目，尤其包括`wx.Bitmap`对象。

对于`wxPython`应用程序，关于所有`UI`的更新，后台线程只负责发送消息给`UI`线程，而不关心`GUI`的更新。幸运的是，`wxPython`没有强制限定你能够有的后台线程的数量。

在这一节，我们将关注几个`wxPython`中实现多线程的方法。最常用的技术是使用`wx.CallAfter()`函数，一会我们会讨论它。然后，我们将看一看如何使用`Python`的队列对象来设置一个并行事件队列。最后，我们将讨论如何为多线程开发一个定制的解决方案。

使用全局函数wx.CallAfter()

例18.5显示了一个使用线程的例子，它使用了 `wxPython` 的全局函数 `wx.CallAfter()` ，该函数是传递消息给你的主线程的最容易的方法。 `wx.CallAfter()` 使得主线程在当前的事件处理完成后，可以对一个不同的线程调用一个函数。传递给 `wx.CallAfter()` 的函数对象总是在主线程中被执行。

图18.4显示了多线程窗口的运行结果。

图**18.4**

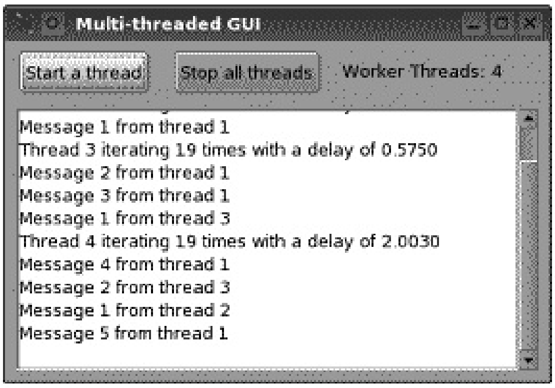


Figure 18.4 Multithreading in the background

例18.5显示了产生图18.4的代码

例**18.5** 使用 **`wx.CallAfter()`** 来传递消息给主线程的一个线程例子

```
#-*- encoding:UTF-8 -*-
import wx
import threading
import random

class WorkerThread(threading.Thread):
    """
 This just simulates some long-running task that periodically se
nds
 a message to the GUI thread.
 """
    def __init__(self, threadNum, window):
        threading.Thread.__init__(self)
        self.threadNum = threadNum
        self.window = window
        self.timeToQuit = threading.Event()
        self.timeToQuit.clear()
        self.messageCount = random.randint(10,20)
        self.messageDelay = 0.1 + 2.0 * random.random()

    def stop(self):
        self.timeToQuit.set()

    def run(self):#运行一个线程
        msg = "Thread %d iterating %d times with a delay of %1.4
f\n" \
              % (self.threadNum, self.messageCount, self.message
```

```
Delay)
        wx.CallAfter(self.window.LogMessage, msg)

        for i in range(1, self.messageCount+1):
            self.timeToQuit.wait(self.messageDelay)
            if self.timeToQuit.isSet():
                break
            msg = "Message %d from thread %d\n" % (i, self.threa
dNum)
            wx.CallAfter(self.window.LogMessage, msg)
        else:
            wx.CallAfter(self.window.ThreadFinished, self)

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Multi-threaded GUI"
)
        self.threads = []
        self.count = 0

        panel = wx.Panel(self)
        startBtn = wx.Button(panel, -1, "Start a thread")
        stopBtn  = wx.Button(panel, -1, "Stop all threads")
        self.tc = wx.StaticText(panel, -1, "Worker Threads: 00")
        self.log = wx.TextCtrl(panel, -1, "",
                               style=wx.TE_RICH|wx.TE_MULTILINE)

        inner = wx.BoxSizer(wx.HORIZONTAL)
        inner.Add(startBtn, 0, wx.RIGHT, 15)
        inner.Add(stopBtn, 0, wx.RIGHT, 15)
        inner.Add(self.tc, 0, wx.ALIGN_CENTER_VERTICAL)
        main = wx.BoxSizer(wx.VERTICAL)
        main.Add(inner, 0, wx.ALL, 5)
        main.Add(self.log, 1, wx.EXPAND|wx.ALL, 5)
        panel.SetSizer(main)

        self.Bind(wx.EVT_BUTTON, self.OnStartButton, startBtn)
        self.Bind(wx.EVT_BUTTON, self.OnStopButton, stopBtn)
        self.Bind(wx.EVT_CLOSE,  self.OnCloseWindow)

        self.UpdateCount()

    def OnStartButton(self, evt):
        self.count += 1
        thread = WorkerThread(self.count, self)#创建一个线程
        self.threads.append(thread)
        self.UpdateCount()
        thread.start()#启动线程

    def OnStopButton(self, evt):
        self.StopThreads()
        self.UpdateCount()
```

```
    def OnCloseWindow(self, evt):
        self.StopThreads()
        self.Destroy()

    def StopThreads(self):#从池中删除线程
        while self.threads:
            thread = self.threads[0]
            thread.stop()
            self.threads.remove(thread)

    def UpdateCount(self):
        self.tc.SetLabel("Worker Threads: %d" % len(self.threads
))

    def LogMessage(self, msg):#注册一个消息
        self.log.AppendText(msg)

    def ThreadFinished(self, thread):#删除线程
        self.threads.remove(thread)
        self.UpdateCount()

app = wx.PySimpleApp()
frm = MyFrame()
frm.Show()
app.MainLoop()
```

上面这个例子使用了 `Python` 的 `threading` 模块。上面的代码使用 `wx.CallAfter(func, *args*)` 传递方法给主线程。这将发送一个事件给主线程，之后，事件以标准的方式被处理，并触发对 `*func(* args)` 的调用。因些，在这种情况中，线程在它的生命周期期间调用 `LogMessage()` ，并在线程结束前调用 `ThreadFinished()` 。

使用队列对象管理线程的通信

尽管使用 `CallAfter()` 是管理线程通信的最简单的方法，但是它并不是唯一的机制。你可以使用 `Python` 的线程安全的队列对象去发送命令对象给 `UI` 线程。这个 `UI` 线程应该在 `wx.EVT_IDLE` 事件的处理函数中写成需要接受来自该队列的命令。

本质上，你要为线程通信设置一个并行的事件队列。如果使用这一方法，那么工作线程在当它们增加一个命令对象到队列时，应该调用全局函数 `wx.WakeUpIdle()` 以确保尽可能存在在一个空闲事件。这个技术比 `wx.CallAfter()` 更复杂，但也更灵活。特别是，这个机制可以帮助你在后台线程间通信，虽然所有的 `GUI` 处理仍在主线程上。

开发你自己的解决方案

你也可以让你自己的工作线程创建一个 `wxPython` 事件（标准的或自定义的），并使用全局函数 `wx.PostEvent(window, event)` 将它发送给 `UI` 线程中的一个特定的窗口。该事件被添加到特定窗口的未决事件队列中，并且 `wx.WakeUpIdle` 自动被调用。这条道的好处是事件将遍历的 `wxPython` 事件设置，这意味你将自由地得到许多事件处理能力，坏处是你不得不自己管理所有的线程和 `wx.CallAfter()` 函数所为你做的事件处理。

本章小结

1、拖放和剪贴板事件是非常相似的，两者都使用了 `wx.DataObject` 来作为数据格式的媒介。除了可以创建自定义的格式以外，还存在着默认的数据对象，包括文本，文件和位图。在剪贴板的使用中，全局对象 `wx.TheClipboard` 管理数据的传送并代表底层系统的剪贴板。

2、对于拖放操作，拖动源和拖动到的目标一起工作来管理数据传送。拖动源事件被阻塞直到拖动到的目标作出该拖动操作是否有效的判断。

3、类 `wx.Timer` 使你能够设置定时的事件。

4、线程在 `wxPython` 是可以实现的，但时确保所有的 `GUI` 活动发生在主线程中是非常重要的。你可以使用函数 `wx.CallAfter()` 来管理内部线程的通信问题。